# Mathematical Engineering of Deep Learning

**Book Draft**

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

# Contents

**Preface - DRAFT** 3

**1 Introduction - DRAFT** 1
- 1.1 The Age of Deep Learning . . . 1
- 1.2 A Taste of Tasks and Architectures . . . 7
- 1.3 Key Ingredients of Deep Learning . . . 12
- 1.4 DATA, Data, data! . . . 17
- 1.5 Deep Learning as a Mathematical Engineering Discipline . . . 20
- 1.6 Notation and Mathematical Background . . . 23
- Notes and References . . . 25

**2 Principles of Machine Learning - DRAFT** 27
- 2.1 Key Activities of Machine Learning . . . 27
- 2.2 Supervised Learning . . . 32
- 2.3 Linear Models at Our Core . . . 39
- 2.4 Iterative Optimization Based Learning . . . 48
- 2.5 Generalization, Regularization, and Validation . . . 52
- 2.6 A Taste of Unsupervised Learning . . . 62
- Notes and References . . . 72

**3 Simple Neural Networks - DRAFT** 75
- 3.1 Logistic Regression in Statistics . . . 75
- 3.2 Logistic Regression as a Shallow Neural Network . . . 82
- 3.3 Multi-class Problems with Softmax . . . 86
- 3.4 Beyond Linear Decision Boundaries . . . 95
- 3.5 Shallow Autoencoders . . . 99
- Notes and References . . . 111

**4 Optimization Algorithms - DRAFT** 113
- 4.1 Formulation of Optimization . . . 113
- 4.2 Optimization in the Context of Deep Learning . . . 120
- 4.3 Adaptive Optimization with ADAM . . . 128
- 4.4 Automatic Differentiation . . . 135
- 4.5 Additional Techniques for First-Order Methods . . . 143
- 4.6 Concepts of Second-Order Methods . . . 152
- Notes and References . . . 164

**5 Feedforward Deep Networks - DRAFT** 167
- 5.1 The General Fully Connected Architecture . . . 167
- 5.2 The Expressive Power of Neural Networks . . . 173
- 5.3 Activation Function Alternatives . . . 180
- 5.4 The Backpropagation Algorithm . . . 184
- 5.5 Weight Initialization . . . 192

5.6 Batch Normalization . . . 194
5.7 Mitigating Overfitting with Dropout and Regularization . . . 197
Notes and References . . . 203

**6 Convolutional Neural Networks - DRAFT 205**
6.1 Overview of Convolutional Neural Networks . . . 205
6.2 The Convolution Operation . . . 209
6.3 Building a Convolutional Layer . . . 216
6.4 Building a Convolutional Neural Network . . . 226
6.5 Inception, ResNets, and Other Landmark Architectures . . . 236
6.6 Beyond Classification . . . 240
Notes and References . . . 247

**7 Sequence Models - DRAFT 249**
7.1 Overview of Models and Activities for Sequence Data . . . 249
7.2 Basic Recurrent Neural Networks . . . 255
7.3 Generalizations and Modifications to RNNs . . . 265
7.4 Encoders Decoders and the Attention Mechanism . . . 271
7.5 Transformers . . . 279
Notes and References . . . 294

**8 Specialized Architectures and Paradigms - DRAFT 297**
8.1 Generative Modelling Principles . . . 297
8.2 Diffusion Models . . . 306
8.3 Generative Adversarial Networks . . . 315
8.4 Reinforcement Learning . . . 328
8.5 Graph Neural Networks . . . 338
Notes and References . . . 353

**Epilogue - DRAFT 355**

**A Some Multivariable Calculus - DRAFT 357**
A.1 Vectors and Functions in $\mathbb{R}^n$ . . . 357
A.2 Derivatives . . . 359
A.3 The Multivariable Chain Rule . . . 362
A.4 Taylor's Theorem . . . 364

**B Cross Entropy and Other Expectations with Logarithms - DRAFT 367**
B.1 Divergences and Entropies . . . 367
B.2 Computations for Multivariate Normal Distributions . . . 369

**Bibliography 399**

**Index 401**

# 6 Convolutional Neural Networks - DRAFT

While offering generality and versatility, the fully connected feedforward neural networks described in the previous chapter are often too general to be effective on their own right. For many applications, such dense architectures can have too many parameters and are not able to generalize well. This is especially the case when considering vision, sound, or similar data for which the spatial orientation of pixels or features is a key defining attribute. For such data, learned rules associated with certain features often need to be repeated systematically. Convolutional neural networks offer an ability to do so by training convolutional filters that can be applied in a spatially homogenous manner. Such networks yield a significant reduction in the number of trained parameters. Understanding convolutional neural networks requires a grasp of the convolution operation and how it is incorporated in a deep learning architecture together with the concept of channels and additional operations such as max-pooling. This chapter covers the main details of such convolutional neural networks as well as specific convolutional architectures that have by now become standard. As the main application of convolutional neural networks is images, we also outline key tasks of deep learning in image processing applications.

We start the chapter with an overview in Section 6.1 where we introduce general concepts of convolutional neural networks. We first touch filtering in signal processing and then consider a high level view of the VGG19 network as a concrete example. In Section 6.2 we study basics of the convolution operation both in one dimension as well as more generally. Towards that end we relate convolutions to systems theory, to probability distributions, and to multiplication of polynomials. In Section 6.3 we focus on a single convolutional layer. First we motivate such a layer and then focus on details such as padding, stride, and dilation. We then introduce the concept of channels and the way that volume convolutions are carried out. In Section 6.4 we put the pieces together and discuss how multiple convolutional layers, and other layers such as max-pooling and fully connected layers, are combined into a network model. Here again, the VGG19 serves as a complete example. In Section 6.5 we describe common landmark architectures and key ideas of convolutional neural networks. The ideas and architectures surveyed include inception networks, ResNets, as well as ways for interpreting the meaning of internal features of the networks. We close with Section 6.6 where we discuss the various tasks that one may consider for vision problems beyond classification. These tasks include object localization, face identification, segmentation models, and others.

## 6.1 Overview of Convolutional Neural Networks

*Convolutional neural networks* (CNNs) are designed to handle *grid-structured data*, such as image data, where there is a strong local dependency between the neighboring items of the grid. For instance, in an image, there is a high chance that adjacent pixels carry similar properties. Such a grid based structure is also present in many sequential data formats such

as text and sound, where a strong correlation exists among adjacent items. Even though this chapter as well as most of the literature on convolutional neural networks focuses on image data, these networks are suitable for working with any temporal, spatial, or spatiotemporal data.

Convolutional neural networks are computationally more efficient than the fully connected neural networks studied in Chapter 5 and are more suitable for grid-structured data. This is primarily because convolutional neural networks require fewer parameters than fully connected networks, with a parameter structure focused on feature (pixel) locality. For instance, consider a classification task for detecting cats within a dataset consisting of images of different animals. Such data exhibits two key properties:

***Translation invariance***: The classification decision for each image is independent of the position of the animal on the image. A cat is a cat irrespective of whether it appears at the top or at the bottom of an image.

***Locality***: The classification decision does not really depend on a pixel that is far away from the animal on the image. A cat is still a cat irrespective of whether far away pixels correspond to a building or a tree.

Ideally, we want our neural network to take advantage of these two properties. When using the fully connected neural networks of Chapter 5 for images, the first step is to convert each image to an input features vector. By doing this we may lose both the above mentioned structural properties of images. On the other hand, convolutional neural networks retain and exploit these properties while generally using fewer parameters.

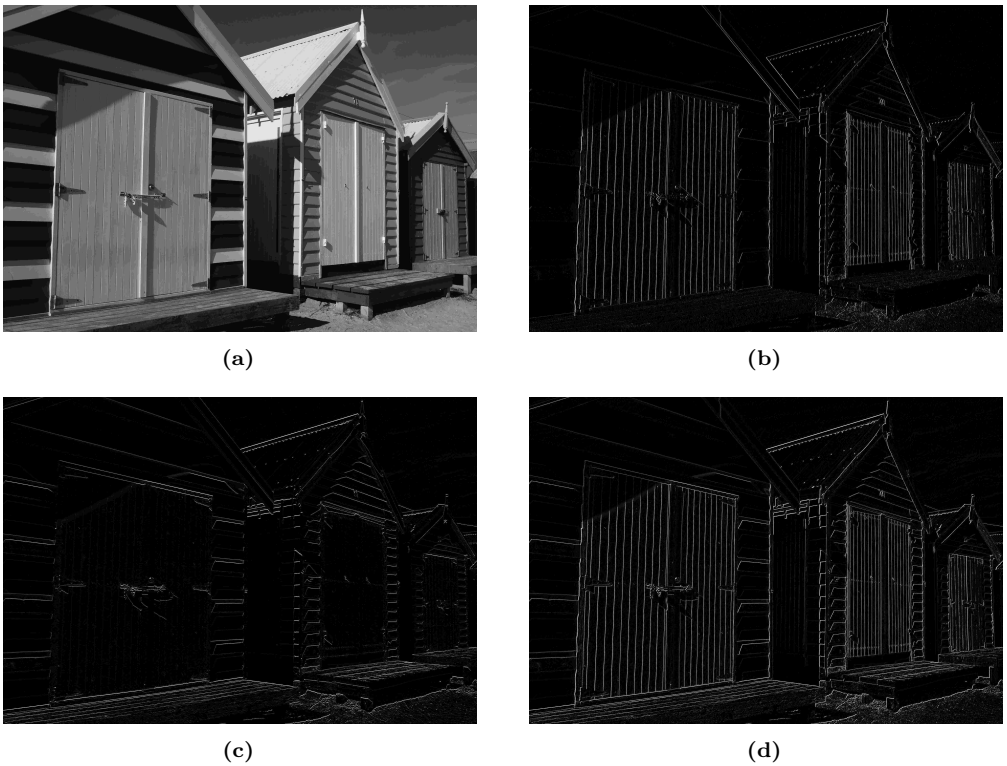
## Filtering

To understand convolutional neural networks it is helpful to have a basic understanding of *filtering*, a well-established field in signal processing, and particularly in the subfields of image processing and computer vision. Filtering is a method or process that removes certain unwanted information from a signal or an image, or alternatively enhances it by accentuating certain information. Taking image processing as an example domain, filtering applies mathematical operations on input images, with the most common operation being the *convolution*. A convolution can be viewed as an operation between two mathematical objects, such as two matrices or two functions, where one object represents an image and the other a filter. All of Section 6.2 is devoted to a basic introduction of convolutions.

In the field of signal processing, each filter is often custom designed depending on the specific task at hand. For example, a popular filter, called the *Sobel filter*, is useful for the task of edge detection. Figure 6.1 illustrates filtering for extracting edges in an image using two Sobel filters applied on an input image appearing in (a). In (b) we detect the vertical edges and in (c) we detect the horizontal edges. By adding the outputs of these two filtering operations, we get the final image shown in (d) which captures most of the vertical and horizontal edge information. More details on edge detection using Sobel filters are in Section 6.2. Beyond edge detection, there are several other tasks that traditional image processing filters can offer, including blurring, smoothing, sharpening, and accentuating images, and each of these is achieved using a custom designed filter.

Convolutional neural networks build upon the classic ideas of filtering using *convolutional layers*. Each convolutional layer is made up of one or more *filters*, also known as *kernels*,

each of which aims to extract a particular feature of the input to the layer. Early layers of the network usually aim to detect lower-level features such as edge detection while the latter layers focus on higher-level features such as identifying cats, dogs, cars, etc. The final hidden layer, ultimately, provides a summary of the input image. Then for example, when considering classification tasks, the final hidden layer is used to classify the input image into different classes.



**(a)** **(b)** **(c)** **(d)**

**Figure 6.1:** Edge detection using the Sobel filter.

The filtering operation at each layer in a convolutional neural network is similar to classical filtering illustrated with edge detection above. However, unlike classical filtering, filters in the convolutional neural network are learned rather than designed. A training dataset is used for learning the filters before using the network for image processing. Here, learning a filter means learning the entries of the matrix that represents the filter, and these entries are called *weights* as the filters play a similar role to the weight matrices of a fully connected neural networks, covered in the previous chapter.

## An Example: VGG19

To get a feel for convolutional neural networks let us consider the task of classifying color images. As an example assume input images are of dimension $3 \times 224 \times 224$, where 3 is the number of channels (red, green, and blue), and $224 \times 224$ specifies the pixel dimensions. Hence the number of input features is $p = 3 \times 224 \times 224 \approx 150,000$. Assume we wish to use such networks for classification of $K = 1,000$ possible classes. If we are to use a fully connected network with no hidden layer, as in the multinomial regression model of Section 3.3, we

already use $p \times K + K$ learned parameters. This is an order of 150 million parameters. Further, deeper networks that extend the multinomial regression model by adding more layers as in Chapter 5, generally require even more parameters. Yet limiting the number of parameters in any machine learning model is important since it bounds computational time, limits usage of computational resources, and reduces overfitting while respecting training data limitations. We now explore what can be done with a convolutional neural network for such a task using approximately the same number of parameters.

The $3 \times 224 \times 224$ input dimensions agree with inputs of the VGG19[1] model, first touched upon in Section 1.1. The VGG19 model has about 144 million parameters (similar to the dense $p \times K + K$ multinomial regression case). However, in contrast to the dense single layer model, VGG19 spans 19 trainable layers! This depth makes the model much more expressive; see Section 5.2 for a discussion on the benefits of depth in networks. Indeed, convolutional neural networks such as VGG19 are specifically suited for image tasks and have a relatively low number of parameters which allow them to be deep.

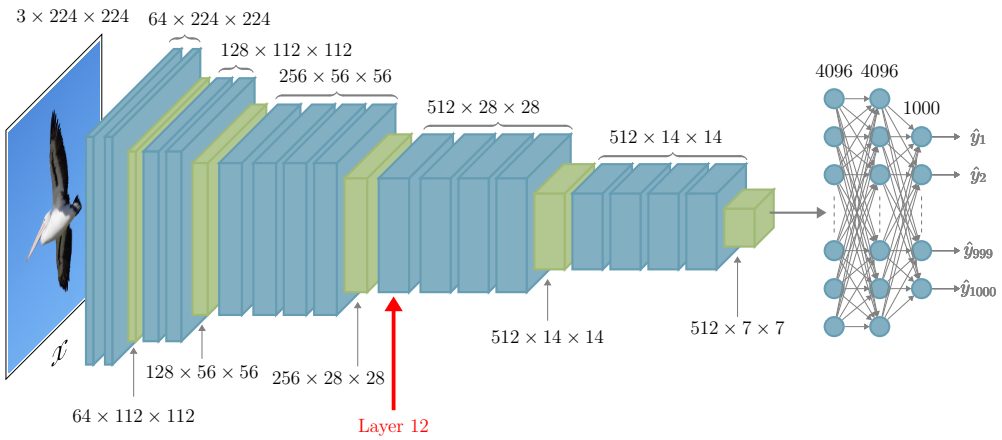



**Figure 6.2:** The VGG19 network architecture. An input $x$ is a $3 \times 224 \times 224$ color image. It is processed through a series of convolutional layers followed by fully connected layers. The resulting output, $\hat{y}_1, \ldots, \hat{y}_{1000}$ is a vector of probabilities indicating the class of the image.

While Sections 6.2, 6.3, and 6.4, introduce the components of convolutional neural networks in detail, at this point let us informally explore the VGG19 model illustrated in Figure 6.2. Like the feedforward networks of Chapter 5, it is composed of layers where data flows between layers down stream from the input $x$ to the output $\hat{y}$. However, unlike feedforward fully connected layers, most layers are not composed of a dense matrix multiplication as in equation (5.2) of Chapter 5, but are rather made of filtering operations implemented via a combination of convolutions and non-linear activation functions. Only the final layers are dense layers.

The rectangular boxes in Figure 6.2 represent neurons, also known as *internal features*, that are computed via the successive application of convolutional layers. Each such box, is in a sense an image or a tensor, yet unlike the input with 3 channels, these internal representations generally have a different number of channels (also known as *feature maps*),

[1] VGG stands for *Visual Geometry Group*, the group at Oxford University that created the network.

not directly corresponding to color values but rather to internal features. As an example consider the layer $\ell = 12$, pointed at with a red arrow in the figure. That layer has 512 channels each containing a $28 \times 28$ pixel "image".

The network also incorporates operations called max-pooling without learned parameters, discussed in detail in Section 6.3. These operations are generally used to reduce dimensions as data flows downstream in the network. Importantly, and quite characteristically of convolutional networks, the VGG19 network starts with a succession of convolutional layers with interleaved max-pooling operations, and towards the end, has dense layers that are similar to the layers of feedforward networks of chapter 5.

## 6.2 The Convolution Operation

The convolution operation is a key component of convolutional neural networks. In this section we study convolutions via various mathematical and engineering viewpoints. We consider linear time invariant systems, probability distributions, multiplication of polynomials, and the general representation of a convolution as a linear operation. We then consider multi-dimensional convolutions and focus on an engineering filtering example, the Sobol filter, used above in Figure 6.1.

A convolution can be viewed as an operation on two functions which creates a third function. In finite domains, these functions may be represented as vectors, matrices, or tensors. We begin the presentation by focusing on one dimensional convolutions. Suppose $f, g : \mathbb{Z} \to \mathbb{R}$ are two functions (or sequences) with discrete domains. Then the *convolution* between $f$ and $g$ is defined as

$$(f \star g)(t) = \sum_{\tau \in \mathbb{Z}} f(t - \tau) g(\tau), \quad t \in \mathbb{Z}. \tag{6.1}$$

In other words, the convolution $f \star g$ between $f$ and $g$ at a point $t$ is obtained by taking summation of the product of the two functions after one of them is *flipped* at the origin and then *shifted* by $t$. The convolution is *commutative*, namely, $(f \star g)(t) = (g \star f)(t)$. This property can be easily observed via a change of variables in the summation of (6.1).

In case $f$ and $g$ have continuous domains, say $f, g : \mathbb{R} \to \mathbb{R}$, the convolution between $f$ and $g$ is defined as

$$(f \star g)(t) = \int_{\mathbb{R}} f(t - \tau) g(\tau) d\tau, \quad t \in \mathbb{R}. \tag{6.2}$$

In both the discrete convolution (6.1) and the continuous convolution (6.2) we assume that the summation or integral, respectively, converges. In the context of deep learning we focus on convolutions of vectors, matrices, and tensors, in which case (6.2) is used on a finite domain and hence always converges. We now present a few viewpoints of one dimensional convolutions before stepping up to multi-dimensional cases.

### Convolutions in LTI Systems

To get a feel for the importance of convolutions we consider *Linear Time Invariant* (LTI) systems. These objects are key in classic control theory and signal processing, and they have influenced machine learning, eventually leading to the development of convolutional neural

networks. An LTI system, denoted here by $\mathcal{L}(\cdot)$, maps an input signal $x = \{x(t) : t \in \mathbb{R} \text{ or } \mathbb{Z}\}$ to an output signal $y$ via $y = \mathcal{L}(x)$. LTI systems satisfy the linearity and time invariance properties:

***Linearity***: For any two input signals $x_1(t)$ and $x_2(t)$ and scalars $\alpha_1$ and $\alpha_2$,

$$\mathcal{L}(\alpha_1 x_1 + \alpha_2 x_2) = \alpha_1 \mathcal{L}(x_1) + \alpha_2 \mathcal{L}(x_2).$$

***Time invariance***: When the shifted (delayed by $\tau$) signal $\tilde{x}(t) = x(t - \tau)$ is given as an input, then the corresponding output signal $\tilde{y} = \mathcal{L}(\tilde{x})$ is $\tilde{y}(t) = y(t - \tau)$, where $y = \mathcal{L}(x)$. Namely, the output of the shifted input is the shifted output of the original input.

An important input signal to consider for any LTI system is the *impulse signal*. In the discrete time case, the impulse signal, denoted by $\delta(t)$, takes 1 at $t = 0$ and 0 for any other $t$, that is,

$$\delta(t) = \begin{cases} 1, & \text{if } t = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Now, the output of the system when the input is the impulse signal is called the *impulse response* and is denoted here as $w = \mathcal{L}(\delta)$. It turns out that the operation of any LTI system on an any input signal $x$ is equivalent to a convolution of $x$ with the impulse response $w$. That is, $\mathcal{L}(x) = w \star x$.

To see this, using this impulse function, any signal $x = \{x(t) : t \in \mathbb{Z}\}$ can be represented as,

$$x(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)\delta(t - \tau),$$

where observe that $\delta(t - \tau)$ takes 1 at $t = \tau$ and 0 otherwise. Consequently, $y(t)$ is equal to,

$$\mathcal{L}(x)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)\mathcal{L}(\delta(t - \tau)) = \sum_{\tau=-\infty}^{\infty} x(\tau)\mathcal{L}(\delta)(t - \tau) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t - \tau) = (w \star x)(t),$$

where the first and second equalities respectively follow from the linearity and the time invariance properties of LTI systems.

A similar result exists for continuous time LTI systems, where the impulse response is the output of the system when the input is a generalized impulse function, $\delta(t)$, called the *Dirac delta function*.[2] Generally, convolutional neural networks do not rely on such continuous time representations. Nevertheless, we mention it here for completeness because most treatments of LTI systems use the delta function.

[2]The Dirac delta function is not an $\mathbb{R} \to \mathbb{R}$ function in the standard sense but is rather a generalized function. It is a mathematical abstraction which allows one to describe an object, $\delta(t)$, that satisfies $\delta(t) = 0$ for $t \neq 0$ as well as $\int_{-\infty}^{\infty} \delta(t)\, dt = 1$. No such standard $\mathbb{R} \to \mathbb{R}$ function exists, but the formalism of generalized functions allows us to treat $\delta(t)$ as though it was standard function. Conceptually, we may also consider $\delta(t)$ as the limit of a Gaussian density centered at zero, with the standard deviation approaching zero.

## Convolutions in Probability

Convolutions also appear naturally in the context of probability. This is when considering the distribution of a random variable that is a sum of two independent random variables. For example consider $\xi = \xi_1 + \xi_2$ for two discrete valued independent random variables $\xi_1$ and $\xi_2$ with probability mass functions $f_1(t)$ and $f_2(t)$, respectively. Then manipulating the probabilities and noting that $\mathbb{P}(A \mid B)$ is the conditional probability of event $A$ given event $B$, we have,

$$\begin{aligned}
\mathbb{P}\big(\xi = t\big) &= \mathbb{P}\left(\xi_1 + \xi_2 = t\right) \\
&= \sum_{\tau=-\infty}^{\infty} \mathbb{P}\big(\xi_1 = \tau\big)\mathbb{P}\big(\xi_1 + \xi_2 = t \;\big|\; \xi_1 = \tau\big) \\
&= \sum_{\tau=-\infty}^{\infty} \mathbb{P}\big(\xi_1 = \tau\big)\mathbb{P}\big(\xi_2 = t - \tau\big) \\
&= \sum_{\tau=-\infty}^{\infty} f_1(\tau) f_2(t-\tau) \\
&= (f_1 \star f_2)(t).
\end{aligned}$$

In other words, the probability mass function[3] of $\xi$ is equal to the convolution of the probability mass functions of $\xi_1$ and $\xi_2$.

## Multiplication of Polynomials and the Convolution Matrix

The convolution also arises when multiplying polynomials. Consider two example polynomials $f(r) = f_0 + f_1 r + f_2 r^2$, and $g(r) = g_0 + g_1 r + g_2 r^2 + \ldots + g_5 r^5$, and their product polynomial $z(r) = f(r)g(r)$. The degree of $z(r)$ is $2 + 5 = 7$ with coefficients $z_0, \ldots, z_7$, as follows,

$$\begin{array}{lcllcl}
z_0 &=& f_0 g_0, & z_4 &=& f_0 g_4 + f_1 g_3 + f_2 g_2, \\
z_1 &=& f_0 g_1 + f_1 g_0, & z_5 &=& f_0 g_5 + f_1 g_4 + f_2 g_3, \\
z_2 &=& f_0 g_2 + f_1 g_1 + f_2 g_0, & z_6 &=& f_1 g_5 + f_2 g_4, \\
z_3 &=& f_0 g_3 + f_1 g_2 + f_2 g_1, & z_7 &=& f_2 g_5.
\end{array} \tag{6.3}$$

With these expressions it is evident that, if we were to set $f_t = 0$ for $t \notin \{0, 1, 2\}$ and similarly $g_t = 0$ for $t \notin \{0, 1, 2, 3, 4, 5\}$, then we could denote,

$$z_t = \sum_{\tau=-\infty}^{\infty} f_\tau g_{t-\tau} = (f \star g)_t.$$

Further, it is also useful to consider the following alternative finite sum representation of $z_t$ given by,

$$z_t = \sum_{i+j=t} f_i g_j,$$

where the sum is over $(i, j)$ pairs with $i + j = t$ and further requiring $i \in \{0, 1, 2\}$ and $j \in \{0, 1, 2, 3, 4, 5\}$. In this context we can also view $z$ as a vector of length 8, $f$ as a vector

[3] Analogous results exist for continuous random variables where probability density functions are used in place of probability mass functions and a continuous convolution such as (6.2) is applied.

of length 3 and $g$ as a vector of length 6. We can then create an $8 \times 6$ *Toeplitz matrix*[4] $T(f)$ that encodes the values of $f$ such that $z = T(f)g$. More specifically,

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} = \underbrace{\begin{bmatrix} f_0 & 0 & 0 & 0 & 0 & 0 \\ f_1 & f_0 & 0 & 0 & 0 & 0 \\ f_2 & f_1 & f_0 & 0 & 0 & 0 \\ 0 & f_2 & f_1 & f_0 & 0 & 0 \\ 0 & 0 & f_2 & f_1 & f_0 & 0 \\ 0 & 0 & 0 & f_2 & f_1 & f_0 \\ 0 & 0 & 0 & 0 & f_2 & f_1 \\ 0 & 0 & 0 & 0 & 0 & f_2 \end{bmatrix}}_{T(f)} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \end{bmatrix}.$$

This shows that the convolution of $f$ with $g$ is a linear transformation given by the matrix-vector product $T(f)g$. Since convolutions are commutative operations, we can also represent the output $z$ as $z = T(g)f$ where $T(g)$ is an $8 \times 3$ Toeplitz matrix. In this case,

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} = \underbrace{\begin{bmatrix} g_0 & 0 & 0 \\ g_1 & g_0 & 0 \\ g_2 & g_1 & g_0 \\ g_3 & g_2 & g_1 \\ g_4 & g_3 & g_2 \\ g_5 & g_4 & g_3 \\ 0 & g_5 & g_4 \\ 0 & 0 & g_5 \end{bmatrix}}_{T(g)} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}.$$

At this point, having seen that convolutions may be encoded via Toeplitz matrices such as $T(f)$ or $T(g)$, we see that the convolution operation is a linear operation. The same also holds for multi-dimensional generalizations which we discuss now.

## Multi-dimensional Generalizations

The convolution operation (6.1) can be generalized to multivariate functions. In fact, for deep learning, convolutions are almost always multivariate. Suppose $f, g : \mathbb{Z}^d \to \mathbb{R}$ are two multivariate functions with discrete domains. Then the *convolution* between $f$ and $g$ is a commutative operation given by

$$(f \star g)(u) = \sum_{v \in \mathbb{Z}^d} f(u - v)\, g(v) = \sum_{v \in \mathbb{Z}^d} f(v)\, g(u - v), \quad u \in \mathbb{Z}^d. \tag{6.4}$$

This is a direct extension of (6.1) with the shifting and the flipping of the functions carried out across all dimensions. Also, similarly to the univariate case over a continuous domain, shown in (6.2), multivariate convolutions have continuous domain representations and these are not presented here because convolutional neural networks use discrete domain convolutions.

The applications presented above for univariate convolutions, namely LTI systems, addition of independent random variables, and multiplication of polynomials, also extend to multivariate

[4] This is a matrix with constant values on the diagonals. Observe that an $n \times m$ Toeplitz matrix requires at most $n + m - 1$ parameters.

cases. Specifically, the probability law of the sum of two independent random vectors can be obtained via a convolution, the coefficients of the product of multivariate polynomials can be obtained via a convolution, and the action of an LTI system operating on a multivariate input signal can be represented via a convolution. This last case is particularly important for this chapter since one often considers a multivariate convolution.

Any vector, matrix, or tensor can be seen as a function from $\mathbb{Z}^d$ to $\mathbb{R}$ with $d = 1$ for vectors, $d = 2$ for matrices, and $d \geq 3$ for general tensors. As a result, the *convolution* between two vectors, two matrices, or two tensors respectively returns a third vector, matrix, or tensor. In particular for the $d = 2$ case, suppose $W = [w_{i,j}]$, for $i = 1, \ldots, K_h$ and $j = 1, \ldots, K_v$, is a $K_h \times K_v$ matrix and $x = [x_{i,j}]$, for $i = 1, \ldots, M_h$ and $j = 1, \ldots, M_v$, is an $M_h \times M_v$ matrix.[5] In this scenario, $f$ and $g$ in (6.4) can be seen as functions from $\mathbb{Z}^2$ to $\mathbb{R}$ by using the matrices $W$ and $x$ respectively by assigning zeros outside the range of their indices. Then we denote the convolution $f \star g$ as $W \star x$. By ignoring obvious zeros in $W \star x$, we can represent this convolution as a matrix of dimension

$$(M_h + K_h - 1) \times (M_v + K_v - 1). \tag{6.5}$$

To see how such output dimensions arise, refer to the analogy in the one dimensional polynomial multiplication (6.3), where we consider an example with input dimensions 3 and 6 (for second and fifth degree polynomials), and thus the output dimension is $3 + 6 - 1 = 8$, matching a 7th degree polynomial.

While (6.5) describes the dimensions of such classical convolutions, the convolution operation used in deep learning differs. In convolutional neural networks, the dimensions of one matrix are smaller than the corresponding dimensions of the other matrix; namely, $K_h \leq M_h$ and $K_v \leq M_v$. In this special case, taking the dimension of $W$ as $K_h \times K_v$ and the dimension of $x$ as $M_h \times M_v$, the convolution $W \star x$ is usually defined to be a matrix of dimension

$$(M_h - K_h + 1) \times (M_v - K_v + 1), \tag{6.6}$$

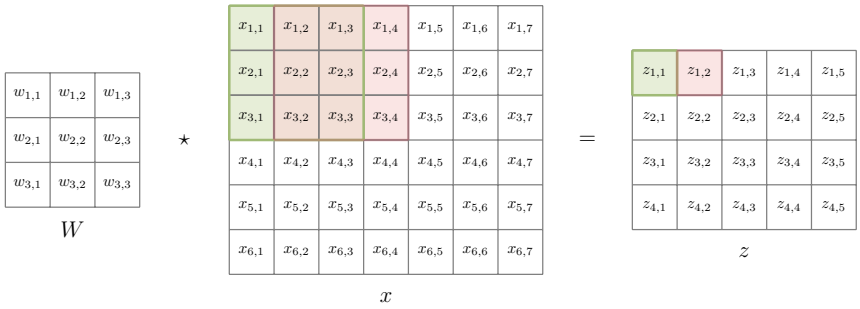
where now for output at $i' = 1, \ldots, M_h - K_h + 1$ and $j' = 1, \ldots, M_v - K_v + 1$, the convolution action is,

$$z_{i',j'} = (W \star x)_{i',j'} = \sum_{i=0}^{K_h-1} \sum_{j=0}^{K_v-1} w_{K_h-i,\, K_v-j}\; x_{i'+i,\, j'+j}, \tag{6.7}$$

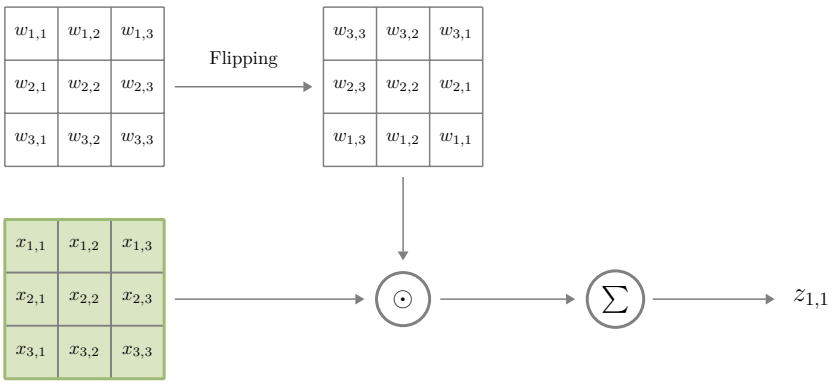
Observe that the convolution in (6.7) is a submatrix of the result one would get if applying (6.4). Further note that in this case, $\star$ is not a commutative operation. Figure 6.3 (a) presents a schematic of the convolution operation (6.7) where $W$ is of dimesnion $K_h \times K_v = 3 \times 3$ and $x$ is of dimesion $M_h \times M_v = 6 \times 7$. Here the output $z = W \star x$ is a $4 \times 5$ dimensional matrix according to (6.6).

The green entry $z_{1,1}$ in Figure 6.3 (a) is based on the green values in $x$ and all of $W$. In more detail, Figure 6.3 (b) presents the computation of the first element $z_{1,1}$. For this we consider the flipped $W$ to obtain another $3 \times 3$ matrix and then take the element-wise product with the sub-matrix of dimension $3 \times 3$ at the top left corner on the matrix $x$ shown in (a) or also shown in green in (b). Similarly, in (a), the red entry $z_{1,2}$ is obtained by sliding the window to the right by one pixel on the matrix $x$ to consider the next $3 \times 3$ sub-matrix (denoted in red).

[5] We use the subscripts $h$ and $v$ in $(M_h, M_v)$ or $(K_h, K_v)$ throughout this chapter. These subscripts stand for 'horizontal' (rows) and 'vertical' (columns) respectively.




**(a)**




**(b)**

**Figure 6.3:** (a) Convolution between two matrices $W$ and $x$ to get $z = W \star x$. The dimensions of $W$ and $x$ are $K_h \times K_v = 3 \times 3$ and $M_h \times M_v = 6 \times 7$, respectively. The dimension of the output $z$ is $(M_h - K_h + 1) \times (M_v - K_v + 1) = 4 \times 5$. (b) Computation of the first element $z_{1,1}$ of the convolution $W \star x$. Here, $\odot$ denotes the element-wise product between two matrices of the same dimension and $\sum$ denotes the summation of all the elements of a matrix.

The convolution operation continues with this process until we reach the top right corner on $x$ to obtain the last element $z_{1,5}$ of the first row of $z$. To obtain the second row, $z_{2,1}, \ldots, z_{2,5}$, we repeat the same process by moving the window one row down on $x$. Ultimately, after the window is at 4 horizontal positions and 5 vertical positions the $4 \times 5$ dimensional output $z$ is obtained. Note that from an implementation perspective, especially when using graphical processing units (GPUs), the computation of $z$ can be parallelized by carrying out the operations illustrated in Figure 6.3 (b) simultaneously for each output $z_{i,j}$.

Now suppose $W$ and $x$ are 3-dimensional tensors with respective dimensions $K_c \times K_h \times K_v$ and $M_c \times M_h \times M_v$, where similarly to before $W$ is smaller than $x$ in the sense that $K_c \leq M_c$, $K_h \leq M_h$, and $K_v \leq M_v$. Here the new dimension sizes $K_c$ and $M_c$ are referred to as the *depth* of the corresponding tensor. For instance, if $x$ denotes a color image then the depth $M_c = 3$ is attributed to the red, blue, and green components of the image. In this 3-dimensional setup, (6.7) can be generalized to provide a *volume convolution*, $W \star x$, with

output dimension,

$$(M_c - K_c + 1) \times (M_h - K_h + 1) \times (M_v - K_v + 1). \tag{6.8}$$

Here, for $k' = 1, \ldots, M_c - K_c + 1$, $i' = 1, \ldots, M_h - K_h + 1$, and $j' = 1, \ldots, M_v - K_v + 1$,

$$(W \star x)_{k',i',j'} = \sum_{k=0}^{K_c-1} \sum_{i=0}^{K_h-1} \sum_{j=0}^{K_v-1} W_{K_c-k,\, K_h-i,\, K_v-j}\; x_{k'+k,\, i'+i,\, j'+j}. \tag{6.9}$$

For deep learning, an important special scenario is the case in which the depths of both $W$ and $x$ are the same, namely $K_c = M_c$. In this case the depth is also called the number of *input channels*. In such a scenario, the dimensions in (6.8) have a depth of 1 and thus the output of the volume convolution $W \star x$ defined by (6.9) can be viewed as a matrix of dimension (6.6). This convolution can also be represented as,

$$W \star x = \sum_{i=1}^{K_c} W_{(i)} \star x_{(i)}, \tag{6.10}$$

where the $\star$ on the right hand side denotes the matrix convolution as in (6.7) and the summation is element-wise. Here the matrices that are convolved are $W_{(i)}$ which is the $i$th matrix along the depth of $W$ and $x_{(i)}$ which is the $i$th matrix along the depth of $x$ (also called the $i$'th input channel).

## Edge Detection Revisited

From an engineering viewpoint, convolutions implement filters, and in the context of image processing (of monochrome images) these are often two dimensional convolutions. We now explore the operation of one such engineered filter, the Sobel filter for edge detection, first mentioned in Section 6.1 and applied in Figure 6.1.

Suppose an input image $x = [x_{i,j}]$ is of dimension $M_h \times M_v$. As we have seen earlier, edge detection involves two separate operations, namely, vertical edge detection and horizontal edge detection, exemplified in Figure 6.1 (b) and (c) respectively. With Sobel filtering, each of these operations is a convolution of $x$ with a $3 \times 3$ kernel matrix given by either,

$$W^{(\leftrightarrow)} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \qquad \text{or} \qquad W^{(\updownarrow)} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix},$$

for horizontal or vertical edge detections, respectively. The actual entries of $W^{(\leftrightarrow)}$ and $W^{(\updownarrow)}$ are part of the Sobel filter design and were engineered[6] to achieve edge detection. Such filters were developed via engineering intuition, trial and error, and experimentation. From our perspective the actual entries of $W^{(\leftrightarrow)}$ and $W^{(\updownarrow)}$ are merely an example since in convolutional neural networks, the values of filters (weights in convolutional layers) are automatically learned during training.

Suppose $y^{(\leftrightarrow)}$ and $y^{(\updownarrow)}$ are the outputs corresponding to horizontal edge detection and vertical edge detection, respectively. These outputs are each computed using (6.7) with $W$

[6] Sobel filters work by approximating the gradient of the image intensity via a discrete differentiation operation.

replaced by $W^{(\leftrightarrow)}$ and $W^{(\updownarrow)}$ respectively, both using the same input image $x$. The overall edge detection can be obtained by superimposing the two outputs as the pixel-wise sum $y^{(\leftrightarrow)} + y^{(\updownarrow)}$, or average $(y^{(\leftrightarrow)} + y^{(\updownarrow)})/2$. In case of color images, one may apply Sobel filters seperatly on each color component, or seek other generalizations and use the convolution formulas (6.9) or (6.10).

## 6.3 Building a Convolutional Layer

In Chapter 5, we have seen the construction of general fully connected neural networks, each of which consists of a series of layers where every neuron in a given layer is connected to every neuron in the next layer. These networks are general in the sense that they are *structure agnostic*, that is, there are no specific assumptions made about the structure of the input. This property makes fully connected neural networks versatile. However, they are inadequate when dealing with specific applications, such as image classification, where the input has rich structural properties.

Convolutional neural networks make use of the aforementioned two key properties of grid-structured data, namely *translation invariance* and *locality*. As a result, the number of parameters to learn in convolutional neural networks is significantly smaller than that of corresponding fully connected neural networks. Convolutional layers are based on the convolution operation, and in this section we focus on building a single convolutional layer.

### Motivating a Convolutional Layer

Convolutional neural networks are designed so that the spatial properties of the image data are inherited from one layer to the next. Therefore, for image processing, it is better to represent both the input and output of a convolutional layer as images. As we are familiar from Chapter 5 with fully connected neural networks, to build a convolutional layer, we begin with a fully connected layer and then we show how the number of learned parameters is reduced using translation invariance and locality.

Consider an input dataset consisting of two dimensional grey scaled images $x$ of dimension $M_h^{[0]} \times M_v^{[0]}$. For the time being, we focus on the first hidden layer of this fully connected network and the superscript $[0]$ denotes that $x$ is an input to this layer. Each input image $x$ is a matrix with the $(i,j)$-th element denoting the pixel value at the $(i,j)$-th location on the image. When treating $x$ as an input to a fully connected neural network, it is represented as an $M_h^{[0]} \cdot M_v^{[0]}$ dimensional vector consisting of all the elements of $x$. Since such a matrix to vector conversion is executed in a consistent manner,[7] without loss of generality we can continue to index the elements of the vector $x$ via tuples $(i,j) \in \{1,\ldots,M_h^{[0]}\} \times \{1,\ldots,M_v^{[0]}\}$.

We wish to represent the output of the first layer also as an image,[8] in this instance having dimension $M_h^{[1]} \times M_v^{[1]}$. Thus, as with the input, the output vector $a^{[1]}$ can also be represented as a matrix, indexed by tuples $(i',j') \in \{1,\ldots,M_h^{[1]}\} \times \{1,\ldots,M_v^{[1]}\}$. As described in Chapter 5, the output $a^{[1]}$ is composed of an affine transformation of $x$ parameterized by $W^{[1]}$ and $b^{[1]}$ composed with a non-linear activation function $S^{[1]}(\cdot)$; see (5.2). Here, with our image based indexing we represent each element of $W^{[1]}$ as $w^{[1]}_{(i',j'),(i,j)}$ and each element

[7] This can be in column major or row major form, and the specific choice between the two is insignificant as long as consistency is maintained.

[8] This requires a non-prime number of neurons in the first layer.

of $b^{[1]}$ as $b^{[1]}_{i',j'}$. With this notation, the output of the layer is

$$a^{[1]} = S^{[1]}(z^{[1]}), \qquad \text{where} \qquad z^{[1]}_{i',j'} = \sum_{(i,j)} w^{[1]}_{(i',j'),(i,j)}\, x_{i,j} + b^{[1]}_{i',j'}. \tag{6.11}$$

It is useful to represent each element of $z^{[1]}$ slightly differently. For this fix $(i', j')$ and reindex the terms in the summation by setting $(i'', j'')$ for each $(i, j)$ such that

$$i = i' + i'', \quad \text{and} \quad j = j' + j''.$$

Now $z^{[1]}_{i',j'}$ can be represented as

$$z^{[1]}_{i',j'} = \sum_{(i'',j'')} w^{[1]}_{(i',j'),(i'+i'',j'+j'')}\, x_{i'+i'',j'+j''} + b^{[1]}_{i',j'}, \tag{6.12}$$

where in the summation, $(i'', j'') \in \{1 - i', \ldots, M^{[0]}_h - i'\} \times \{1 - j', \ldots, M^{[0]}_v - j'\}$. Observe that generally these indices, $i''$ and $j''$, take on both positive and negative values as they reflect the offset relative to $i'$ and $j'$ respectively.

We now return to the first structural property of image data, namely, translation invariance. With this property, we expect that any shift in $x$ results only as a shift in the output. As an illustration, let us revisit edge detection and consider a pelican in flight as shown in Figure 6.4. In (a) we see an input to an edge detection filter and in (b) we have the output. Similarly, (c) and (d) are input–output pairs of a similar image. Observe that the pairs (a)-(b) and (c)-(d) are essentially the same, except for the fact that the position of the pelican in the output depends only on its position in the input. In other words, the filtering operation's action on the object is generally independent of the location of the object in the image.

In mathematical terms, such translation invariance implies that the weights $w^{[1]}_{(i',j'),(i'+i'',j'+j'')}$ must be independent of the output indices $(i', j')$ because $(i', j')$ is the pixel location in the output image. With the change of variables, we can use $i''$ and $j''$ as relative offsets to that pixel coordinate instead of absolute coordinates. We can then define a smaller set of parameters made of weights $w_{i'',j''}$ and a scalar bias $b$ such that for all output coordinates $(i', j')$, the original parameters are

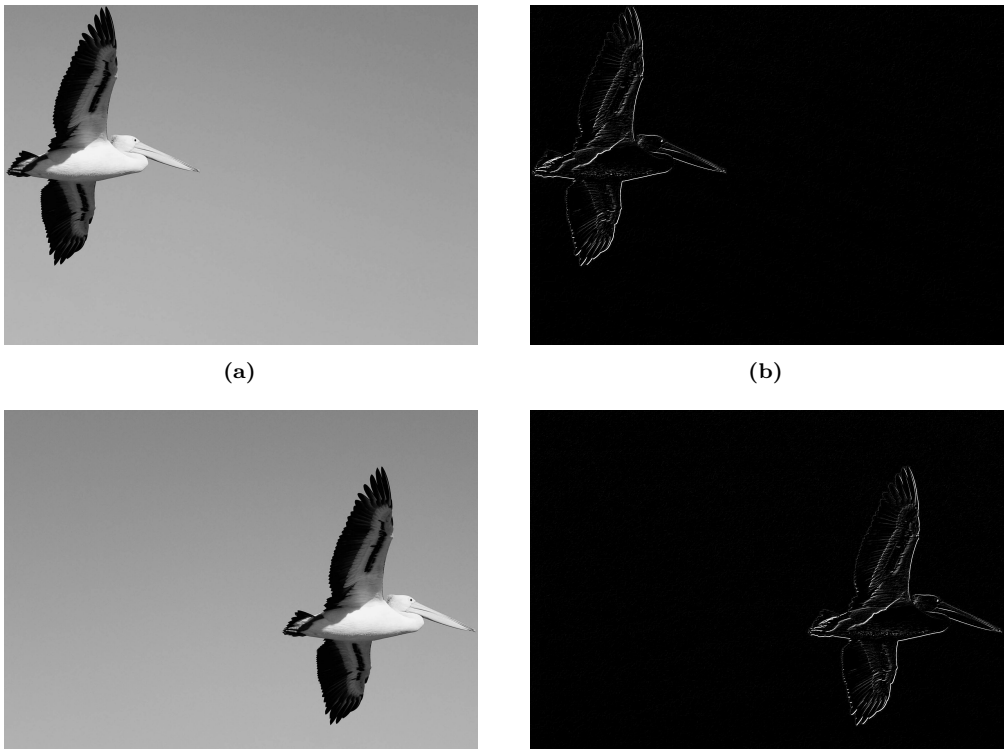
$$w^{[1]}_{(i',j'),(i'+i'',j'+j'')} = w_{i'',j''} \qquad \text{and} \qquad b^{[1]}_{i',j'} = b.$$

This simplifies the expression for $z^{[1]}_{i',j'}$ in (6.12) to be,

$$z^{[1]}_{i',j'} = \sum_{(i'',j'')} w_{i'',j''}\, x_{i'+i'',j'+j''} + b. \tag{6.13}$$

The expression (6.13) already indicates a significant reduction in the number parameters to learn in comparison to (6.12). To see this observe that in (6.12) our weights potentially vary based on $i'$ and $j'$ whereas in (6.13) they do not.

We now see further reduction of the parameters by invoking the second structural property, namely locality. Viewed in terms of pixels, this property states that a pixel $x_{i,j}$ is not significantly influenced by far away pixels. A motivational illustration is in Figure 6.5



**(a)** **(b)**

**(c)** **(d)**

**Figure 6.4:** Edge detection of images with a pelican to illustrate the property of translation invariance.

consisting of pelicans and seagulls, with each individual bird enclosed in a red box. Generally, the structural property of locality implies that if we are seeking information about one of these specific birds, then it is sufficient to know the pixel information only within the box that is enclosing the bird. Similarly, at a much finer level when we seek information about edges or similar features, it is often enough to consider 1, 2, or 3, neighboring pixels in each direction – yielding convolution kernels of size $3 \times 3$, $5 \times 5$, or $7 \times 7$ respectively.

**(a)** **(b)**

**Figure 6.5:** Images of birds to illustrate the property of locality. The pixel information within each red box is typically sufficient for understanding the characteristics of the bird inside the box.

To mathematically enforce locality for the evaluation of $z^{[1]}_{i',j'}$, we ignore the pixel values $x_{i'+i'',j'+j''}$ for $i'' < 0$, $j'' < 0$, $i'' \geq K_h$, and $j'' \geq K_v$ for some chosen $K_h, K_v > 0$; e.g. $K_h, K_v$ at 3, 5, or 7. Equivalently, we set $w_{i'',j''} = 0$ for all $(i'', j'')$ with $i'' \notin \{0, \ldots, K_h - 1\}$ and $j'' \notin \{0, \ldots, K_v - 1\}$. This further reduces the layer's affine transformation to be,

$$z^{[1]}_{i',j'} = \underbrace{\sum_{i''=0}^{K_h-1} \sum_{j''=0}^{K_v-1} w_{i'',j''}\, x_{i'+i'',j'+j''}}_{\star} + b, \tag{6.14}$$

where the first term marked by $\star$ is essentially a convolution $W \star x$ with $W$ denoting a kernel matrix[9] of dimension $K_h \times K_v$. Hence, the operation of the layer can be represented as

$$a^{[1]} = S^{[1]}(z^{[1]}), \qquad \text{where} \qquad z^{[1]} = (W \star x) + b, \tag{6.15}$$

where the addition of the scalar bias $b$ is element wise to each element of the matrix $W \star x$. Note that the $\star$ convolution operation in (6.14) and (6.15) is slightly different from (6.7) studied in the previous section. To see this difference, recall that the $(i', j')$-th element of the convolution operation (6.7) is given by

$$\sum_{i''=0}^{K_h-1} \sum_{j''=0}^{K_v-1} w_{K_h-i'',K_v-j''}\, x_{i'+i'',j'+j''},$$

and compare this with the summation marked by $\star$ in (6.14). Hence in our case, $W \star x$ is the conventional convolution if we replace each $w_{i'',j''}$ with $w_{K_h-i'',K_v-j''}$; i.e., flipping at the origin. In the context of neural networks, such a replacement only implies reindexing of the learned parameters and has no effect on the network structure or its performance. For instance, if we observe the edge detection operation illustrated in Figure 6.1, the filter $w$ is flipped only once, and after that for any input $x$ we obtain an element-wise product between the flipped $w$ and sub-matrices of $x$. Therefore, learning a filter and learning its flipped version are equivalent. As a result, in deep learning, the flipping operation is avoided for simplicity. In any case, the kernel matrix $W$ is still called a *convolutional kernel*.

In summary we have seen that at its core, a single convolutional layer involves the following actions on the input $x$. First it is convolved with a convolution kernel $W$. Then the result is shifted by a scalar bias $b$. Finally an activation function $S^{[1]}(\cdot)$ is applied. These actions are summarized in (6.15). Note that when the input dimension is $M^{[0]}_h \times M^{[0]}_v$, using (6.6), the dimension of the output is,

$$M^{[1]}_h \times M^{[1]}_v = (M^{[0]}_h - K_h + 1) \times (M^{[0]}_v - K_v + 1). \tag{6.16}$$

For an illustration of the reduction in the number of parameters that a convolutional layer has in comparison to a fully connected layer, consider an example with input dimension $M^{[0]}_h \times M^{[0]}_v = 224 \times 224$ and a case with kernel dimension $K_h \times K_v = 3 \times 3$. Here with (6.16), the output dimension is $M^{[1]}_h \times M^{[1]}_v = 222 \times 222$. If we were to seek the same size of output dimension with a fully connected layer, we have $222 \times 222 = 49,284$ neurons. Since the input size is $224 \times 224 = 50,176$, the dimension of the weight matrix is the product of the input size and output size (number of neurons), and together with the bias vector (one entry

[9] In Chapter 5 the notation $W$ is used for weight matrices whereas here it is a (generally) smaller kernel matrix. Note that it implicitly defines a weight matrix, not directly used in computation.

for each neuron) we have $2,472,923,268$ parameters. In contrast, in the convolutional layer there are only $3 \times 3 + 1 = 10$ parameters. While on its own, such a single convolutional layer is certainly not as expressive as the fully connected layer with 2.5 billion learned parameters, as we see below, combining convolutional layers in tandem yields very powerful networks with much fewer parameters than their fully connected counterparts.

## Alterations to the Convolution: Padding, Stride, and Dilation

The convolution appearing in (6.14) is often tweaked and modified in the context of image data. Specifically, alterations to the convolution operation, known as *padding*, *stride*, and *dilation*, are sometimes employed. For a fixed kernel of dimension $K_h \times K_v$, the combination of these modifications allows us to control the output size as well as the effective input size. Before diving into the details, we mention that these alterations are parameterized by non-negative integer pairs, $(p_h, p_v)$ for padding, $(s_h, s_v)$ for stride, and $(d_h, d_v)$ for dilation, where the subscript $h$ is for height and the subscript $v$ is for width.

In the basic convolution operation above, the absence of padding, stride, and dilation is via a selection of $(0, 0)$ for padding and stride, as well as a selection of $(1, 1)$ for dilation. Such a choice yields output dimension as in (6.16). However, when increasing these integers (typically by small single digit numbers), the output dimension formula (6.16) is generalized to,

$$M_h^{[1]} \times M_v^{[1]} = \left(1 + \left\lfloor \frac{M_h^{[0]} - d_h(K_h - 1) - 1 + p_h}{s_h} \right\rfloor\right) \times \left(1 + \left\lfloor \frac{M_v^{[0]} - d_v(K_v - 1) - 1 + p_v}{s_v} \right\rfloor\right), \tag{6.17}$$

where $\lfloor u \rfloor$ represents the largest integer not greater than $u$. We now introduce and motivate each of these alterations separately and develop (6.17). The reader may verify that with the aforementioned default settings (0 for padding and stride, and 1 for dilation), (6.17) reduces to (6.16).

To motivate padding, recall the edge detection example above. Due to the convolution operation, the output image dimension is smaller than the input image dimension. In particular, since the filter dimension $K_h \times K_v$ is $3 \times 3$ (Sobel filter), when the input dimension is $M_h^{[0]} \times M_v^{[0]}$, the output dimension is equal to $(M_h^{[0]} - 2) \times (M_v^{[0]} - 2)$ as in (6.16). Hence we see a slight reduction of the image size at the output. Since convolutional neural networks typically consist of several convolutional layers, the dimension reductions in each of these layers can accumulate, making the overall downstream dimension undesirably small. *Padding* is a simple solution to overcome this problem by adding extra zero-valued pixels around the input so that the effective input dimension is higher, and the desired output dimension is obtained.

To illustrate padding consider the example in Figure 6.3 (a). Here a convolutional layer with a kernel of dimension $3 \times 3$ is applied to inputs of dimension $6 \times 7$. Without padding, for each input we get an output of dimension $4 \times 5$. Now suppose we increase the dimension of the input to $8 \times 9$ by adding zeros around the input image. Then when we apply the convolution on the modified input, the output dimension is $6 \times 7$, which is equal to the unpadded input image dimension. Figure 6.6 illustrates this operation.

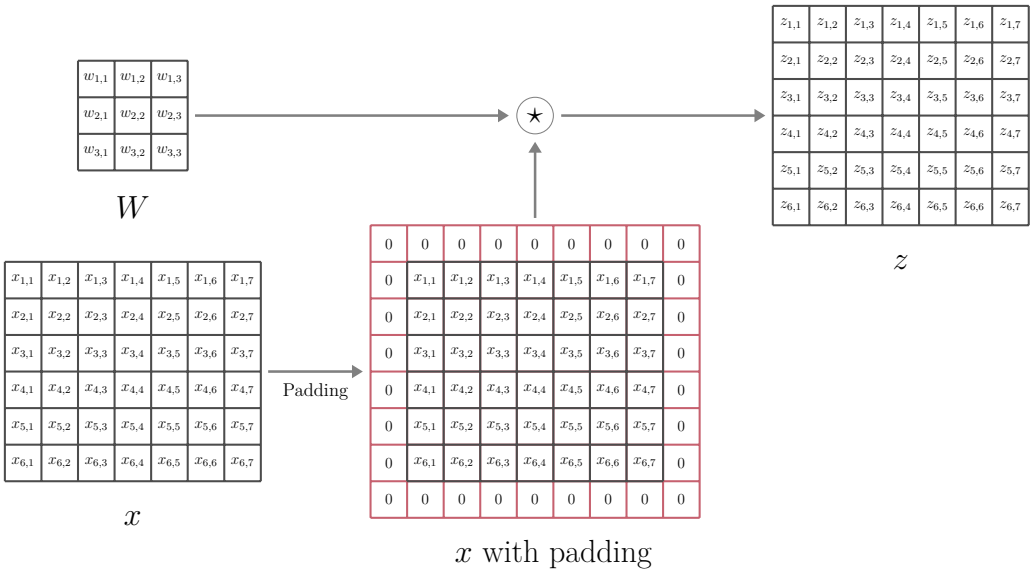



**Figure 6.6:** Illustration of convolution with padding. In this example a $3 \times 3$ convolution with a padding setting of $(p_h, p_v) = (2, 2)$ maintains the same dimensions for the output $z$ as the input $x$.

More generally, again suppose that the input dimension is $M_h^{[0]} \times M_v^{[0]}$ and the kernel dimension is $K_h \times K_v$. Further suppose that each input image is modified by adding $p_h$ rows roughly half on the top and half on the bottom, and $p_v$ columns roughly half on the left and half on the right. Then it is easy to check that (6.16) is modified so that the output dimension is

$$M_h^{[1]} \times M_v^{[1]} = (M_h^{[0]} - K_h + p_h + 1) \times (M_v^{[0]} - K_v + p_v + 1). \tag{6.18}$$

Note that setting $(p_h, p_v) = (K_h - 1, K_v - 1)$ is a mechanism for ensuring that the input and the output are of the same dimension. Also note that typically convolutional neural networks are designed to have kernels of odd height and odd width. Hence it is common to pad with exactly $p_h/2$ rows of zeros on the left and $p_h/2$ rows of zeros on the right, and similarly for the vertical dimension as shown in (6.6). This helps maintain spatial symmetry while conducting convolutions.

The convolutions we presented up to now involved shifts of the convolution kernel by one pixel at a time. This is called a convolution with a *stride of one*, or $(s_h, s_v) = (1, 1)$. However, in many applications, we may wish to slide the convolution kernel with bigger steps in order to either reduce the computational cost, or to reduce the dimension of the output of the convolutional layer. This is achieved by adjusting the stride size $(s_h, s_v)$ to be greater than one.

As a toy example consider Figure 6.7 where the dimension of the input is $10 \times 10$ (potentially after padding), and the kernel is of dimension $3 \times 3$. For this example let us use an hypothetical stride setting of $(s_h, s_v) = (5, 4)$. This setting implies that the convolution kernel is shifted in each step with 5 pixels down, or 4 pixels to the right. As usual we start from the top-left

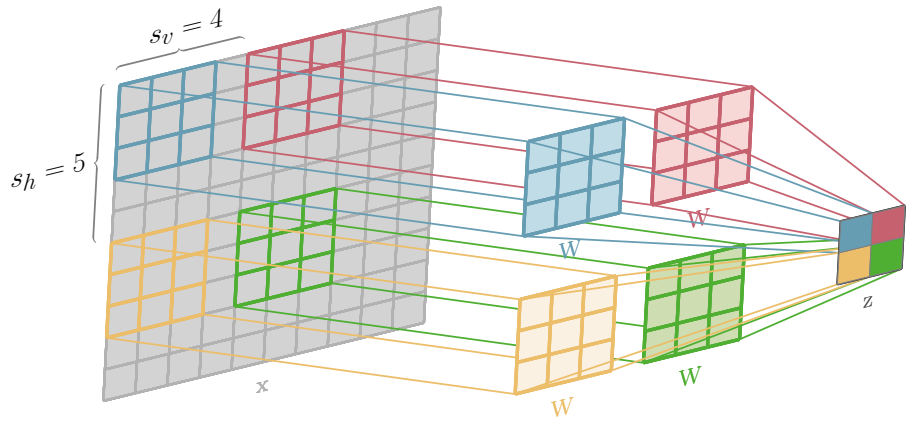



**Figure 6.7:** Illustration of a convolution with stride settings $(s_h, s_v) = (5, 4)$. In this hypothetical example there is no overlap, yet in practice one often uses smaller stride settings.

corner, placing the $3 \times 3$ convolution kernel on the input image to compute the first element of the output. After computing each element, we move the convolution kernel by 4 pixels to the right and compute the next element of the row. Once a row of the output is finished, we move the convolution kernel downwards by 5 pixels and repeat the horizontal shifting for the next row of the output. Each time we compute an element of the output, we make sure there are enough selected input pixels for the convolution kernel.

Note that in this example, for ease of presentation in the figure, we chose stride settings greater than the size of the convolution kernel and this implies no overlap of the sliding windows. However, in practice one typically uses stride settings of size 2, 3, or similar small steps, smaller than $K_h$ and $K_v$, and this yields overlap in the convolution multiplications. In general the effect of a stride is in data reduction allowing us to create outputs that are smaller in dimensions than the input, yet capture the essential information. A second mechanism for such reductions is pooling, a concept described in Section 6.4.

The alternation of convolutions, with stride settings $(s_h, s_v)$, modifies the output dimension equation from (6.18) to

$$M_h^{[1]} \times M_v^{[1]} = \left(1 + \left\lfloor \frac{M_h^{[0]} - K_h + p_h}{s_h} \right\rfloor\right) \times \left(1 + \left\lfloor \frac{M_v^{[0]} - K_v + p_v}{s_v} \right\rfloor\right). \tag{6.19}$$

The expression results from the fact that the number of elements computed in each row of the output after computing the first element of the row is equal to the number of rightwards moves allowed. With an effective input row size of $M_v^{[0]} + p_v$, this number is $\lfloor (M_v^{[0]} - K_v + p_v)/s_v \rfloor$. Adding 1 due to the first element, yields the width of the output; similarly for the height.

We now focus on *dilation*, a technique for increasing the *receptive field*. The receptive field of an individual filter is marked by the dimensions of the window in the input $x$ that affect a single pixel in the output. For example with a standard $3 \times 3$ convolution, the receptive

field is $3 \times 3$ since each pixel in $W \star x$ is influenced by a $3 \times 3$ window in $x$. When layers are composed, the receptive field has a more general meaning since as data propagates down the network, the receptive field grows.

Dilation increases the respective field by spreading out the elements of the kernel matrix $W$ via the insertion of zeros between elements. This alteration allows the kernel to cover a larger area of the input image without increasing the number of learned parameters. The level of dilation is determined by the settings $(d_h, d_v)$ where dilation converts a kernel of size $K_h \times K_v$ to a kernel of size $K'_h \times K'_v = (d_h(K_h - 1) + 1) \times (d_v(K_v - 1) + 1)$. Specifically, dilation adds $d_h - 1$ all-zero rows between each pair of adjacent rows from the original kernel, and similarly adds $d_v - 1$ columns. Thus the number of all zero rows added is $(K_h - 1)(d_h - 1)$, and similarly $(K_v - 1)(d_v - 1)$ for columns. See Figure 6.8 for an example with $(d_h, d_v) = (2, 2)$.

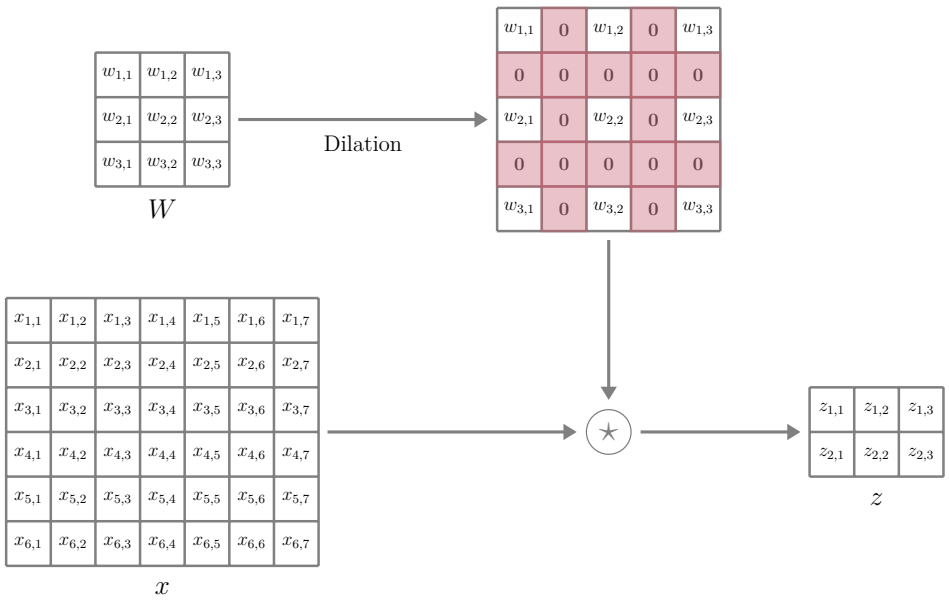



**Figure 6.8:** Illustration of dilation operation with $(d_h, d_v) = (2, 2)$ extending a $3 \times 3$ convolution filter to create a receptive field of $5 \times 5$.

Overall, together with a padding of size $(p_h, p_v)$ and a stride of size $(s_h, s_v)$, a dilation factor of $(d_h, d_v)$ implies that the output dimension is determined by (6.17). To see this, replace $K_h$ and $K_v$ in (6.19) with the effective kernel sizes $K'_h$ and $K'_v$, respectively.

## Inputs with Multiple Channels

So far in this section we have looked at the case where each input is a matrix, usually representing a grey scale image. However, convolutional networks often deal with inputs comprised of multiple *channels*. For instance, a color image has three channels representing the red, green, and blue components. When we have such data with multiple channels, input to a convolutional layer is no longer a matrix but is rather represented as a three dimensional tensor. We denote this tensor's dimensions via $M_c^{[0]} \times M_h^{[0]} \times M_v^{[0]}$, where the *depth* $M_c^{[0]}$

denotes the number of channels, and the other two numbers are for the height and width dimensions as used previously. Hence, for color images we have $M_c^{[0]} = 3$ and further, as we describe in the sequel, for hidden layers we often have more than 3 input channels to the layer.

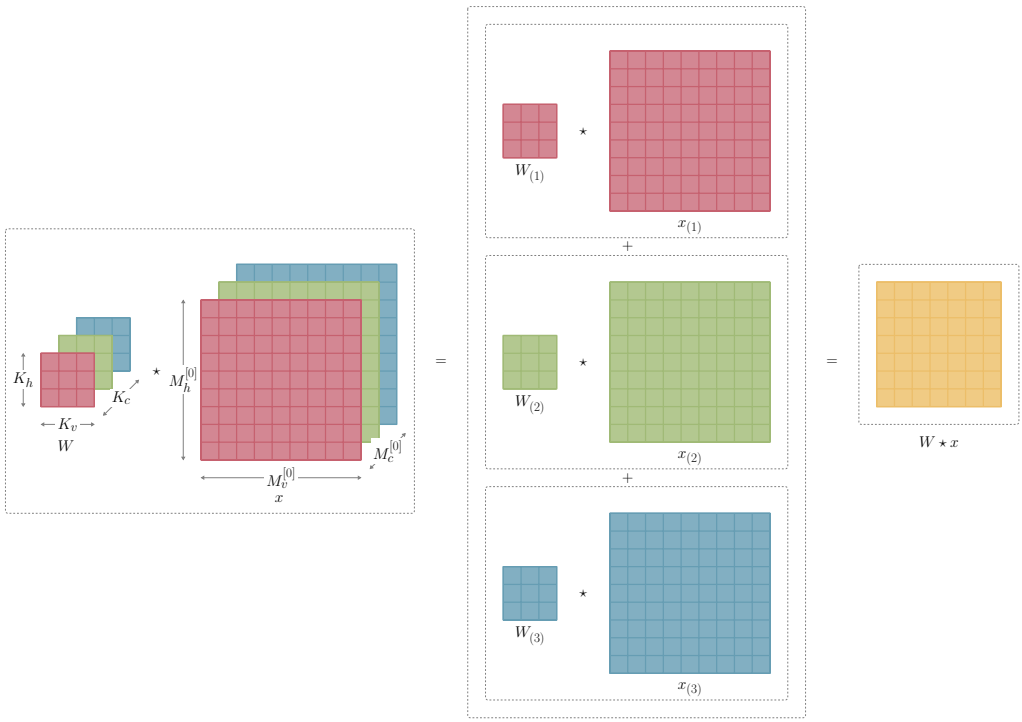



**Figure 6.9:** Graphical representation of the typical convolution operation with $K_c = M_c^{[0]}$. In this example $M_c^{[0]} = 3$ input channels and we use an $M_c^{[0]} \times 3 \times 3$ convolution kernel.

To deal with inputs with multiple channels we often conduct a volume convolution as in (6.9). For this we use a kernel $W$ with depth greater than one which is a three dimensional tensor with dimensions denoted via $K_c \times K_h \times K_v$, such that $K_c \leq M_c^{[0]}$, $K_h \leq M_h^{[0]}$, and $K_v \leq M_v^{[0]}$. In fact, the typical case is to set $K_c = M_c^{[0]}$ where the output is a matrix and the convolution is as in (6.10).

Namely for input tensor $x$, the convolution $W \star x$ is a matrix which is computed via an element-wise sum of the two dimensional convolutions $W_{(i)} \star x_{(i)}$ for $i = 1, \ldots, M_c^{[0]}$. Each $W_{(i)} \star x_{(i)}$ is a matrix of dimension $M_h^{[1]} \times M_v^{[1]}$ as in (6.17). This two dimensional convolution is based on the $i$th channel in the input tensor, denoted $x_{(i)}$, and on $W_{(i)}$ which is the corresponding $K_h \times K_v$ dimensional matrix matching channel $i$ in the convolution kernel tensor $W$. Note that the same settings of padding, stride, and dilation are applied across all the channels. Figure 6.9 illustrates such a volume convolution for the case of $M_c^{[0]} = K_c = 3$.

After the volume convolution is carried out, a single scalar bias term, $b$, is added to each element of the matrix $W \star x$. Then a (generally) non-linear activation function $S^{[1]}(\cdot)$ is applied. Hence the action of the convolution on multiple input channels parallels (6.15) and is,

$$a^{[1]} = S^{[1]}(z^{[1]}), \qquad \text{where} \qquad z^{[1]} = \Big( \sum_{i=1}^{M_c^{[0]}} W_{(i)} \star x_{(i)} \Big) + b. \tag{6.20}$$

## Outputs with Multiple Channels

Until now, regardless of the number of input channels, the output is a matrix, denoted via $a^{[1]}$ in (6.20). This is because, so far there is only one kernel, possibly a tensor, operating on the input to the convolutional layer. However, most popular convolutional neural networks have convolutional layers with multiple kernels operating on the input simultaneously. In this case, the output of the layer is a collection of matrices denoted by $a_{(j)}^{[1]}$ for $j = 1, \ldots, M_c^{[1]}$, where $M_c^{[1]}$ is the *number of output channels* (also known as *feature maps*). Consequently, the output can be viewed as a 3-dimensional tensor of dimension $M_c^{[1]} \times M_h^{[1]} \times M_v^{[1]}$.

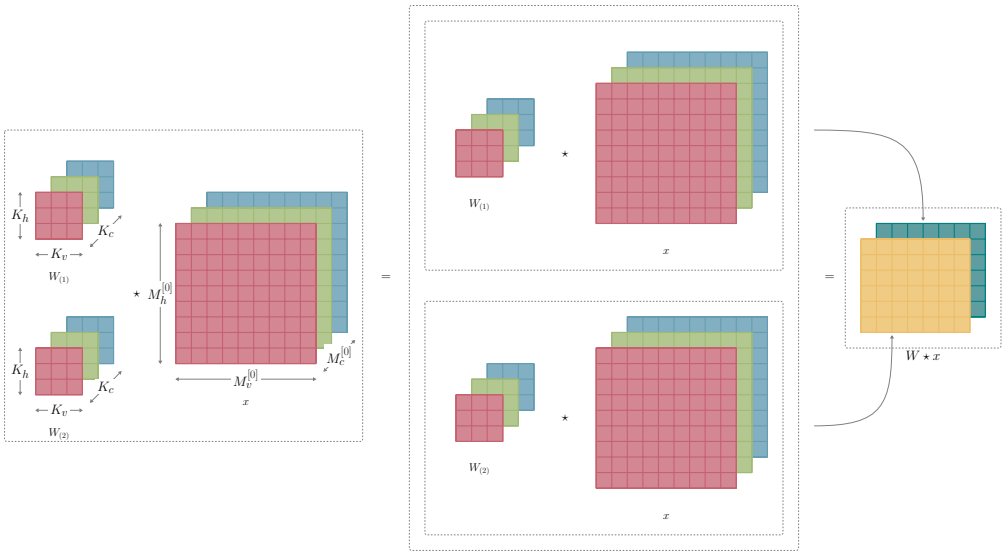



**Figure 6.10:** Illustration of a convolutional layer with 3 input channels and 2 output channels.

In this case, the convolutional layer is parameterized by multiple kernels $W_{(j)}$ for $j = 1, \ldots, M_c^{[1]}$, each with a scalar bias term $b_{(j)}$. In particular, the kernel $W_{(j)}$ and bias term $b_{(j)}$ correspond to the output channel $j$. With this notation, the operation of the layer can be represented as,

$$a_{(j)}^{[1]} = S^{[1]}(z_{(j)}^{[1]}), \qquad \text{where} \qquad z_{(j)}^{[1]} = \Big( \sum_{i=1}^{M_c^{[0]}} W_{(j),(i)} \star x_{(i)} \Big) + b_{(j)}, \tag{6.21}$$

for $j = 1, \ldots, M_c^{[1]}$, where similarly to (6.20), $W_{(j),(i)}$ is the matrix corresponding to the $i$th input channel for the $j$th kernel. See Figure 6.10 for an illustration in the case of $M_c^{[0]} = 3$ and $M_c^{[1]} = 2$ (3 input channels and 2 output channels).

It is a common practice to use the same dimension $K_c \times K_h \times K_v$ for all kernels $W_{(j)}$ of the layer with the same settings of padding $(p_h, p_v)$, stride $(s_h, s_v)$, and dilation $(d_h, d_v)$ for all the channels. In that case, the dimension $M_h^{[1]} \times M_v^{[1]}$ of each output channel is given by (6.17).

As an illustrative hypothetical example of multiple output channels, assume that the input to the first layer is a color image with three channels. One kernel can be used to extract horizontal edges in each input channel of the image while another kernel of the same size extracts vertical edges. In that case, the output has two channels where one consists of horizontal edges and the other consists of the vertical edges. More generally, in trained networks, we can think of different channels of the output as different feature extractions from the input. These channels jointly help in overall feature extraction for the whole network.

## 6.4 Building a Convolutional Neural Network

We have now acquired all the crucial elements necessary for constructing convolutional neural networks, such as the VGG19 model depicted in Figure 6.2. We now put the pieces together for constructing a convolutional neural network that, in addition to convolutional layers, includes fully connected layers, as studied in Chapter 5, and *pooling* layers described in this section. This section also offers complete details of the previously introduced VGG19 network, serving as an illustrative example. It also introduces *fully convolutional networks*, an architecture that uses convolutional layers in place of fully connected layers.

A convolutional neural network is generally deep with multiple layers, similar to feedforward networks studied in Chapter 5. Unlike feedforward networks which consist of only fully connected layers, convolutional neural networks have different types of layers, of which some are trainable and the others are not, and the trainable layers are further broken up into *convolutional layers* and *dense layers*. Using the notation of Chapter 5, we use $L$ for the number of layers, and decompose $L$ to

$$L = L_{\text{train}} + L_{\text{pool}}, \qquad \text{where} \qquad L_{\text{train}} = L_{\text{conv}} + L_{\text{dense}}.$$

Here $L_{\text{train}}$, counts the number of trainable layers, whereas $L_{\text{pool}}$ counts the number of layers that do not have trainable parameters. Further, the trainable layers are either convolutional layers, counted by $L_{\text{conv}}$, or fully connected layers, counted by $L_{\text{dense}}$. It is important to note that in terms of naming conventions, in some instances the depth of the network is taken as $L$, whereas in others it is taken as $L_{\text{train}}$. For example, in the VGG19 network,

$$L = 24, \qquad L_{\text{train}} = 19, \qquad L_{\text{pool}} = 5, \qquad L_{\text{conv}} = 16, \qquad L_{\text{dense}} = 3, \tag{6.22}$$

yet the network is called VGG19 and not "VGG24".

Similar to a feedforward network, the goal of a convolutional neural network is to approximate some unknown function $f^*(\cdot)$. For instance, for classification of image data with animal faces, the function value $f^*(x)$ for any given image $x$ may yield a probability vector with the

highest weight on the index associated with the label of the image $x$. A convolutional neural network defines a mapping $f_\theta(\cdot)$ and learns the values of the unknown parameters $\theta$ that ideally result in $f^*(x) \approx f_\theta(x)$ for as many input images $x$ as possible. In general, similar to equation (5.1) for feedforward networks, the approximating function $f_\theta(\cdot)$ is recursively composed as

$$f_\theta(x) = f^{[L]}_{\theta^{[L]}}(f^{[L-1]}_{\theta^{[L-1]}}(\dots(f^{[1]}_{\theta^{[1]}}(x))\dots)),$$

where for each $\ell$, the function $f^{(\ell)}_{\theta^{[\ell]}}(\cdot)$ is associated with the $\ell$th layer which depends on the layer's parameters $\theta^{[\ell]} \in \Theta^{[\ell]}$. Note that for layers that are not trainable (as counted via $L_{\text{pool}}$), the parameter space $\Theta^{[\ell]}$ is empty.

In general, similarly to feedforward networks, it is useful to denote the neuron activations of the network via $a^{[1]}, a^{[2]} \dots, a^{[L]}$ where $a^{[L]} = \hat{y}$ is the output, and for $\ell = 1, \dots, L-1$,

$$a^{[\ell]} = f^{[\ell]}_{\theta^{[\ell]}}(a^{[\ell-1]}),$$

with $a^{[0]} = x$. We mention that the shape of the neurons per layer $a^{[\ell]}$ varies as it is sometimes a tensor (of order 3) and sometimes a vector, depending on the type of layer.

## Convolutional Layers

When the $\ell$-th layer of the network is a convolutional layer, then $f^{(\ell)}_{\theta^{[\ell]}}(\cdot)$ uses (6.21), treating $a^{[\ell-1]}$ as the input. In this case the input and output are generally 3-tensors as we have seen in the previous sections. In particular,

$$f^{(\ell)}_{\theta^{[\ell]}} : \mathbb{R}^{M_c^{[\ell-1]} \times M_h^{[\ell-1]} \times M_v^{[\ell-1]}} \longrightarrow \mathbb{R}^{M_c^{[\ell]} \times M_h^{[\ell]} \times M_v^{[\ell]}},$$

maps $a^{[\ell-1]}$ of dimension $M_c^{[\ell-1]} \times M_h^{[\ell-1]} \times M_v^{[\ell-1]}$ to $a^{[\ell]}$ of dimension $M_c^{[\ell]} \times M_h^{[\ell]} \times M_v^{[\ell]}$. Now (6.21) is implement for the $M_c^{[\ell]}$ output channels and this operation can be represented as,

$$f^{(\ell)}_{\theta^{[\ell]}}\big(a^{[\ell-1]}\big) = S^{[\ell]}\Big( \underbrace{\Big[ b^{[\ell]}_{(j)} + \sum_{i=1}^{M_c^{[\ell-1]}} W^{[\ell]}_{(j),(i)} \star a^{[\ell-1]}_{(i)} \Big]}_{z^{[\ell]}_{(j)}}{}_{j=1,\dots,M_c^{[\ell]}} \Big),$$

where the input tensor has $M_c^{[\ell-1]}$ channels and the output tensor has $M_c^{[\ell]}$ channels. Using similar notation to (6.21), the kernel $W^{[\ell]}_{(j)}$ is of dimension $K_c^{[\ell]} \times K_h^{[\ell]} \times K_v^{[\ell]}$ (the same for all output channels $j$) where the kernel matrix for the $i$-th input channel and $j$-th output channel is denoted $W^{[\ell]}_{(j),(i)}$. The tensor after the volume convolutions and bias term is denoted using the notation $\big[z^{[\ell]}_{(j)}\big]_{j=1,\dots,M_c^{[\ell]}}$ where each $z^{[\ell]}_{(j)}$ is a matrix of dimension $M_h^{[\ell]} \times M_v^{[\ell]}$.

Note that the activation function $S^{[\ell]}(\cdot)$ is now considered as a function applied on a tensor of dimension $M_c^{[\ell]} \times M_h^{[\ell]} \times M_v^{[\ell]}$. It is typically an element-wise application of scalar activation functions $\sigma^{[\ell]}(\cdot)$ similarly to previous feedforward examples. In fact, the common activation function is $\sigma_{\text{ReLU}}(\cdot)$; see Section 5.3.

Observe that the number of learned parameters for the layer is ,

$$M_c^{[\ell]} \cdot \left( M_c^{[\ell-1]} \cdot K_h^{[\ell]} \cdot K_v^{[\ell]} + 1 \right), \tag{6.23}$$

since there are $M_c^{[\ell]}$ kernels (one per output channel) each of dimension $K_c^{[\ell]} \times K_h^{[\ell]} \times K_v^{[\ell]}$ where $K_c^{[\ell]} = M_c^{[\ell-1]}$ (the number of input channels to the layer) and since each output channel adds a scalar bias term.

## Pooling Layers

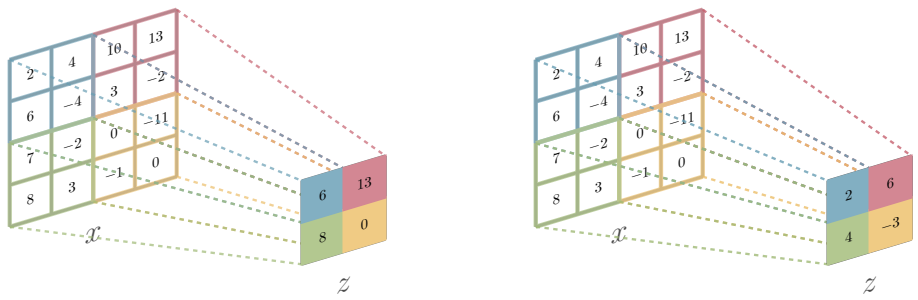
As mentioned above, there are also non-trainable layers counted by $L_{\text{pool}}$ and these are typically pooling layers. The main idea of a pooling layer is to reduce the height and width of the input tensor $a^{[\ell-1]}$ to achieve a lower dimensional output tensor $a^{[\ell]}$ while retaining the same number of channels. The operation of the layer can be summarized with a function,

$$f_{\text{pool}}^{(\ell)} : \mathbb{R}^{M_c^{[\ell-1]} \times M_h^{[\ell-1]} \times M_v^{[\ell-1]}} \longrightarrow \mathbb{R}^{M_c^{[\ell]} \times M_h^{[\ell]} \times M_v^{[\ell]}},$$
$$\text{with} \quad M_c^{[\ell-1]} = M_c^{[\ell]}, \; M_h^{[\ell-1]} > M_h^{[\ell]}, \quad \text{and } M_v^{[\ell-1]} > M_v^{[\ell]}.$$

Generally for some fixed channel $j$, and pixel coordinates of the output $(i, k)$, a pooling operation operates on pixels from a window in the input denoted via $\mathcal{I}_{(i,k)}$. Here $\mathcal{I}_{(i,k)}$ is a set of pixel coordinates in the input that are maped to the specific output pixel $(i, k)$. There are two popular pooling techniques used in practice, namely, *max-pooling* and *average-pooling*. For each channel $j$, the pooling operation can be summarized as,

$$\left[a_{(j)}^{[\ell]}\right]_{i,k} = \begin{cases} \max_{(i',k') \in \mathcal{I}_{(i,k)}} \left[a_{(j)}^{[\ell-1]}\right]_{i',k'}, & \text{(max-pooling)} \\ \frac{1}{|\mathcal{I}_{(i,k)}|} \sum_{(i',k') \in \mathcal{I}_{(i,k)}} \left[a_{(j)}^{[\ell-1]}\right]_{i',k'}. & \text{(average-pooling)} \end{cases}$$

As is evident, max-pooling takes the maximal pixel value within the window as the output, while average pooling averages pixel values within the window for the output.




**(a)** **(b)**

**Figure 6.11:** An example of pooling with a $2 \times 2$ window. (a) Max-pooling. (b) Average-pooling.

The specifics of the pooling operation define exactly how $\mathcal{I}_{(i,k)}$ is determined. Generally, similar to the convolution operation and its alternations with stride and padding, we may view pooling as moving a small window over the input to compute an output. The way in which this window moves implicitly defines $\mathcal{I}_{(i,k)}$. As a concrete example see Figure 6.11 which illustrates a case of pooling with a window of dimensions $2 \times 2$. With this, $|\mathcal{I}_{(i,k)}| = 4$, and then each output pixel $(i, k)$ is computed based on all 4 pixels $(i', k') \in \mathcal{I}_{(i,k)}$ from the input image which form a $2 \times 2$ window in $a_{(j)}^{[\ell-1]}$. A typical *pooling stride* of the window which covers all input pixels while forming non-overlapping windows is to shift each time with the size of the window as in Figure 6.11. In general other pooling stride settings are also possible.

The idea of pooling interplays with the notion that the initial layers of a convolutional network focus on pixel level features similar to edge detection, and as we progress towards the final layers of the network, the information is aggregated to address general questions about the whole image. Thus deeper layers are less sensitive to translation changes on the input image compared to the initial layers. For instance, the answer to a question "is there a bird in the photo?" is the same for both images in Figure 6.5, even though the corresponding outputs from the initial layers look different. Pooling layers are applied after convolutional layers to help achieve such aggregation by reducing the spatial dimension of the outputs. In addition, the dimension reduction which pooling layers offer is important from a computational perspective.

We now return to the notion of a receptive field, previously discussed in the context of dilation and with respect to a single convolution. Now we consider it in the context of a whole network. In particular we consider the *receptive field of a derived feature*. Consider a neuron in the network, $\left[a_{(j)}^{[\ell]}\right]_{i,k}$, for layer $\ell$, channel $j$, and pixel coordinates $(i, k)$. This neuron or activation is a derived feature inside the network. Using the dimensions and specifications of the layers up to that neuron, namely $1, \ldots, \ell$, it can be determined which input pixels from the input image $x$, affect the value of $\left[a_{(j)}^{[\ell]}\right]_{i,k}$. For example if the neuron is at a first layer involving a $5 \times 5$ convolution kernel, then the value of the neuron is only determined by 25 pixels in the input image. However, if the layer $\ell$ is a hidden layers with multiple convolutions and pooling layers prior to it, it may be that $\left[a_{(j)}^{[\ell]}\right]_{i,k}$ is determined by the whole input image $x$ or a significant portion. In general, pooling layers help increase the receptive field of neurons of hidden layers. This allows the derived features towards the end of the network to depend on the whole input image, or significant parts of it.

## Fully Connected Layers

When the $\ell$-th layer of the network is a fully connected layer then the operation of $f_{\theta^{[\ell]}}^{(\ell)}(\cdot)$ is as in (5.2) of Chapter 5. Such layers are typically deployed at the end of the network. This is because the typical task of the last layers of convolutional neural network is to address general questions, such as classification of the objects in the image. Note that since fully connected layers operate on vectors as the input, in cases where the previous layer has a tensor as output, the tensor is flattened to a vector.

It is common to adapt the final fully connected layers of convolutional networks for specific tasks. For example, the VGG19 model can have the final layers fine tuned for tasks such as object localization discussed in Section 6.6. In doing so, we may take the network trained for classification, and then fine tune it for the other task by only training the fully connected

layers. This is sometimes called *freezing layers* (the ones not trained) during training.. Similarly, convolutional networks that were trained on generic images from a general domain, such as ImageNet, can be fine tuned by training the final layers on more specific images from a specific domain (e.g., only on specific animal images of a certain type). This process, also used in other non-convolutional models, is called *transfer learning*.

## VGG19 Revisited

We now take a closer look at the architecture of our running example network, VGG19. While this is not the most modern convolutional architecture, it is instructive to consider it here since it falls directly within the paradigms discussed above. Other popular convolutional architectures are surveyed in the next section. We have seen in (6.22) the counts of different layer types in VGG19 which has $L = 24$ layers of which 19 are trainable. Table 6.1 provides complete details.

Each input to the network is a color image $x$ of dimension $M_c^{[0]} \times M_h^{[0]} \times M_v^{[0]} = 3 \times 224 \times 224$. In the basic form we present here, the network is configured for a classification task with $K = 1,000$ classes. Thus, the output $\hat{y}$ of the network is a probability vector of length $1,000$, where the $i$th element, $\hat{y}_i$, denotes the probability of $x$ is of class $i \in \{1, \ldots, K\}$.

In this architecture all the convolutional kernels in the network are of the same dimension, $K_h \times K_v = 3 \times 3$. The padding and stride settings are the same for all the convolutional layers with $(p_h, p_v) = (2, 2)$ for padding and $(s_h, s_v) = (1, 1)$ for stride. There is no dilation, i.e. $(d_h, d_v) = (1, 1)$. With these settings, it is evident from (6.17) that for each convolutional layer, the input height and width dimensions are identical to the output height and width dimensions. Thus with this network, height and width dimensions are reduced only via pooling. All the pooling layers are max-pooling using $2 \times 2$ windows that are moved with a stride of $(2, 2)$ without padding. Thus each such pooling layer halves the height and width dimensions. The dimensions start at $224 \times 224$ and are halved using the sequence, $224, 112, 56, 28, 14$, and $7$. Yet as layers progress, more channels are added where we start with 3 channels in the input and increase to 64 channels in the first layer. Then after some of the pooling layers, we double the number of channels so that eventually by layer $\ell = 12$ there are 512 channels.

We see from Table 6.1 that the tensor output of the 21st layer, which is a max-pooling layer, is flattened to a vector that is given as an input to the first fully connected layer, layer $\ell = 22$; namely $512 \times 7 \times 7 = 25,088$. In terms of activation function, the architecture uses the Rectified Linear Unit (ReLU) activation function for all the hidden trainable layers and soft-max for the output layer so that each output assigns a probability to each of the possible $1,000$ classes.

In the original VGG19 paper,[10] the network was trained on the ImageNet dataset and nowdays when one uses this network, one often uses a pretrained version. In the original paper the input images were preprocessed by subtracting the mean red, green, and blue value, computed over the entire ImageNet training set, from each pixel. This type of *preprocessing* is needed in production (test time) as well. Note that in the original paper, to obtain the input size $224 \times 224$, input images were randomly cropped from rescaled training images, one

[10] The VGG19 architecture achieved state-of-the-art performance on the ImageNet classification task in 2014, with a top-5 error rate of 7.3%. This network is often used as a pre-trained model for transfer learning tasks, where the lower layers are fixed and the higher layers are fine tuned for a specific task.

**Table 6.1:** Specifications of the VGG19 architecture. The number of learned parameters for the covolutional layers is computed using (6.23). The number of learned parameters for a fully connected layer with input size $N^{[\ell-1]}$ and output size $N^{[\ell]}$ is $N^{[\ell-1]} \cdot N^{[\ell]} + N^{[\ell]}$; see Section (5.1) for more details on the learned parameters of fully connected layers.

| Layer. No. | Type of Layer | Output Dimension | No. of Neurons | No. of Learned Parameters |
|---|---|---|---|---|
| 0 | Input | $3 \times 224 \times 224$ | - | - |
| 1 | Convolution | $64 \times 224 \times 224$ | $3,211,264$ | $1,792$ |
| 2 | Convolution | $64 \times 224 \times 224$ | $3,211,264$ | $36,928$ |
| 3 | Max-pooling | $64 \times 112 \times 112$ | $802,816$ | $0$ |
| 4 | Convolution | $128 \times 112 \times 112$ | $1,605,632$ | $73,856$ |
| 5 | Convolution | $128 \times 112 \times 112$ | $1,605,632$ | 147,584 |
| 6 | Max-pooling | $128 \times 56 \times 56$ | $401,408$ | $0$ |
| 7 | Convolution | $256 \times 56 \times 56$ | $802,816$ | $295,168$ |
| 8 | Convolution | $256 \times 56 \times 56$ | $802,816$ | $590,080$ |
| 9 | Convolution | $256 \times 56 \times 56$ | $802,816$ | $590,080$ |
| 10 | Convolution | $256 \times 56 \times 56$ | $802,816$ | $590,080$ |
| 11 | Max-pooling | $256 \times 28 \times 28$ | $200,704$ | $0$ |
| 12 | Convolution | $512 \times 28 \times 28$ | $401,408$ | $1,180,160$ |
| 13 | Convolution | $512 \times 28 \times 28$ | $401,408$ | $2,359,808$ |
| 14 | Convolution | $512 \times 28 \times 28$ | $401,408$ | $2,359,808$ |
| 15 | Convolution | $512 \times 28 \times 28$ | $401,408$ | $2,359,808$ |
| 16 | Max-pooling | $512 \times 14 \times 14$ | $100,352$ | $0$ |
| 17 | Convolution | $512 \times 14 \times 14$ | $100,352$ | $2,359,808$ |
| 18 | Convolution | $512 \times 14 \times 14$ | $100,352$ | $2,359,808$ |
| 19 | Convolution | $512 \times 14 \times 14$ | $100,352$ | $2,359,808$ |
| 20 | Convolution | $512 \times 14 \times 14$ | $100,352$ | $2,359,808$ |
| 21 | Max-pooling | $512 \times 7 \times 7$ | $25,088$ | $0$ |
| Flattening to a vector of length $25,088$ | | | | |
| 22 | Fully connected | $4,096$ | $4,096$ | $102,764,544$ |
| 23 | Fully connected | $4,096$ | $4,096$ | $16,781,312$ |
| 24 | Fully connected | $1,000$ | $1,000$ | $4,097,000$ |
| | | | Total: 16,391,656 | Total: 143,667,240 |

crop per image per each iteration of the stochastic gradient descent optimization algorithm. This type of data augmentation is further discussed in Chapter 8.

## One by One Convolutions and Fully Convolutional Networks

A *one by one convolutional layer* is a special case of a convolutional layer where we apply $K_h^{[\ell]} \times K_v^{[\ell]} = 1 \times 1$ dimensional kernel matrices on all the input channels. At first glance, if one returns to the basics of two dimensional convolutions as in (6.7), it may seem like a one by one convolution is nothing but a scalar multiplication. However, since now there are $K_c^{[\ell]}$ (or $M_c^{[\ell-1]}$) channels at play, the one by one convolution allows us to create a linear combination of the input channels. For example, in image processing when one converts a red, green, and blue color image into a monochrome (black and white) image, one way to do

so is to define each monochrome pixel as a linear combination of the three color pixel values, and this is a one by one convolution.

One obvious application of one by one convolutions is for the reduction of depth (number of channels) inside convolutional neural networks without changing the spatial dimension. Return to the VGG19 architecture in Table 6.1 and observe that from layer 0 to layer 21 depth either stays the same or grows (starting at 3 and reaching 512). However, in contrast to VGG19 that flattens layer 21, say we wanted to have a layer, which we call a *depth reduction layer*, straight after layer 21, which reduces the depth from 512 channels to a lower number. This can be viewed as a non-linear projection of the 512 channels in layer 21 onto a tensor of lower dimension with less channels. Clearly, one by one convolutions offer a natural way for such depth reduction where we set the number of one by one convolution kernels as the desired number of output channels of the reduction layer. So for example in VGG19 if we would have wanted layer 22 to be a tensor of dimension $8 \times 7 \times 7$ instead of the fully connected layer as in Table 6.1, then we would introduce 8 one dimensional convolutions for that layer. The total parameter count for that layer would be $8 \times 512 + 8 = 4,112$, where the additional $+8$ is for the bias term of each of the 8 one by one convolutions.

Importantly, one by one convolutions also allow us to represent fully connected layers as convolutional layers. To see this, recall that a fully connected layer relies on an affine transformation on some input vector, say $x$ of length $N$, to obtain $z = Wx + b$, where $W$ is the weight matrix with $N$ columns, and $b$ is the bias vector. In that case, the $j$-th element of $z$ is,

$$z_j = \Big( \sum_{i=1}^{N} W_{j,i}\, x_i \Big) + b_j. \tag{6.24}$$

Now return to (6.21) and consider a one by one convolution on a volume $x$ of dimension $N \times 1 \times 1$. In this case $x_{(i)}$ can simply be represented as $x_i$ and the $\star$ operation can be replaced by multiplication. Omitting the superscript "[1]" in (6.21), we have,

$$z_{(j)} = \Big( \sum_{i=1}^{N} W_{(j),(i)} \cdot x_i \Big) + b_{(j)}. \tag{6.25}$$

Hence, we see that the fully connected operation (6.24) and the one by one convolution operation (6.25) are essentially identical.

In general a convolutional network that does not have fully connected layers and has all trained weights and biases associated with convolutional layers is called a *fully convolutional network*. In essence a non fully convolutional network such as VGG19 may be transformed into a fully convolutional network by replacing the fully connected layers using one by one convolutions. This process sometimes termed *convolutionalization*. For example for VGG19 this means transforming layers 22, 23, and 24, as in Table 6.1, into convolutional layers. There are multiple reasons for convolutionalization and multiple advantages to fully convolutional architectures. Primarily, the representation of fully connected layers as convolutional layers allows us to stack multiple parallel outputs or intermediate channels in a single tensor.

## Dropout, Batch Normalization, and Group Normalization

Some of the techniques introduced in Chapter 5 for fully connected neural networks are also applicable in convolutional networks. We now discuss two such techniques, namely dropout

and batch normalization. We also highlight group normalization which is a variant of batch normalization in the context of convolutional neural networks.

Recall from Section 5.7 that *dropout* is a simple regularization technique where during each forward pass in the training, only a random subset of the neurons (randomly selected for that iteration) is used. In convolutional networks, we can still employ dropout for the fully connected layers but not for the convolutional layers. This is because in convolutional layers, the neurons have spatial orientation, and dropping out individual neurons could disrupt the spatial structure.

*Batch normalization*, introduced in Section 5.6, often accelerates learning. The key idea is a shifting and scaling transformation using additional learned parameters as in (5.40) of Chapter 5 which generally maintains the activation values in a dynamic range near 0. For convolutional neural networks, batch normalization at a convolutional layer $\ell$ is usually applied on each channel $z_{(j)}^{[\ell]}$ of the convolution output $z^{[\ell]}$ before the corresponding activation is applied. That is, for two learned scalar parameters $\gamma_j^{[\ell]}$ and $\beta_j^{[\ell]}$, the $j$th channel matrix of dimension $M_h^{[\ell]} \times M_v^{[\ell]}$ after the is given by

$$\tilde{z}_{(j)}^{[\ell]} = \gamma_j^{[\ell]} \bar{z}_{(j)}^{[\ell]} + \beta_j^{[\ell]}, \tag{6.26}$$

for each $j = 1, \ldots, M_c^{[\ell]}$, where as before we use '+' for addition of the scalar to every element of the matrix. Here, the matrix being transformed has $(i', j')$-element $[\bar{z}_{(j)}^{[\ell]}]_{i',j'}$ that is computed similar to (5.39) by subtracting the mean and then dividing it by the square-root of the variance plus a small constant $\varepsilon$, where the mean and variance are computed for the same element $(i', j')$ of $j$th channel of the convolution output $z_{(j)}^{[\ell]}$ over the entire mini-batch, similar to (5.38).

A variant that has gained popularity is called *group normalization*. Here, instead of normalizing each channel (applying (6.26) on a standardized $\bar{z}_{(j)}^{[\ell]}$), the channels of the convolution output are divided into a set of groups, and then the mean and variance values are computed for each group over a mini-batch and similarly a form of (6.26) is applied per-group. Hence the learned parameters ($\gamma$'s and $\beta$'s) are per group in a layer. Note that the group normalization is identical to the batch normalization when the number of groups is equal to the number of channels, but otherwise it reduces the number of learned parameters.

## Understanding Inner Layers and Derived Features

Recall an elementary example from Section 2.2 where we estimated a simple linear regression coefficient $\hat{\beta}_1$ to have a value of 8.27. In that simple example, the *interpretation* of the estimated parameter was clear: A unit increase of the feature implies an increase of the output by 8.27. Thus with linear models, beyond using the model for prediction, the actual learned parameters have meaning. Ideally, for deep learning models in general, and particularly for convolutional neural networks, we would also like to have such an interpretation of the learned parameters. Namely, what information do we know based on the learned convolution kernels, weight matrices, and bias vectors. However, convolutional (and deep) models with

millions of parameters are much more involved, and simple direct interpretability is typically not attainable.[11]

While direct interpretability is not possible, there are multiple techniques for visualizing convolutional neural networks. We now briefly summarize some overarching approaches which we dichotomize as either *weight based* or *feature based*. The weight based approach focuses on visualizing the learned convolution kernels of the network. The feature based approach focuses on the activation values in specific channels and has several variants.

Starting with the *weight based approach*, visualizing the weights of convolution kernels with $K_c$ at most 3 is possible just by treating the kernel as a red, green, and blue image and displaying it. For many architectures this is possible at the first layer since the input has three channels (hence $K_c = 3$). In fact, for many trained models, the color image visualization of first layer convolution kernels shows that these filters are similar in nature to simple engineered filters such as edge detectors. On the other hand, for layers down the network, there are often more than three channels, and while we may try to use data reduction techniques to visualize the associated filters (each with $K_c > 3$), such a visualization is typically not fruitful.

Continuing to the *feature based approach*, we focus on the values of activations in specific channels in the network. A simple mechanism is to apply different categories of images and examine which neurons or activations are most excited by which category. A slightly more sophisticated feature based approach is via the application of *occlusions* (covering part of the view). The basic idea is to first consider a non-occluded image, and then occlude the image by covering up some interesting part such as a face of a person. We then compare the difference in neuron activation values for the non-occluded and occluded inputs. Neurons for which the difference in activation values is significant may then be interpreted as being sensitive to the occluded part of the image (e.g. to a face).

All the aforementioned approaches are simple in the sense that they do not rely on an additional model, but rather just use the trained model under study. However, there are multiple approaches that execute additional optimization for better interpretability insights. As an illustration, let us see one such approach stemming from a landmark paper.[12] In addition to the methodological contribution, the work of this paper also highlighted important structural aspects of trained convolutional neural networks. Specifically, it was shown that initial layers of the network generally seek simple visual features such as corners, colors, and edges, while later layers of the network find much more refined artifacts such as faces, or specific objects.

Consider Figure 6.12 which illustrates a visual interpretation of some channels within a trained convolutional network. The network has many channels across multiple layers, and here we present only a few of those channels, focusing on a pair of arbitrary channels within each of the layers 2, 3, 4, and 5. Before we outline how the visualization in this figure was created, let us interpret it. Each channel that we visualize has a $3 \times 3$ grid of synthesized images (channel visualization) as well as a matching $3 \times 3$ grid of parts of images from a

[11] We mention that there is a whole field dealing with *interpretable machine learning*. In this subsection our goal is to only present a glimpse of the area.

[12] See "Visualizing and Understanding Convolutional Networks" by M. Zeiler and R. Fergus, [441].

[13] Image is adapted from figure 2 of "Visualizing and understanding convolutional networks", [441] with thanks to M. D. Zeiler and R. Fergus.

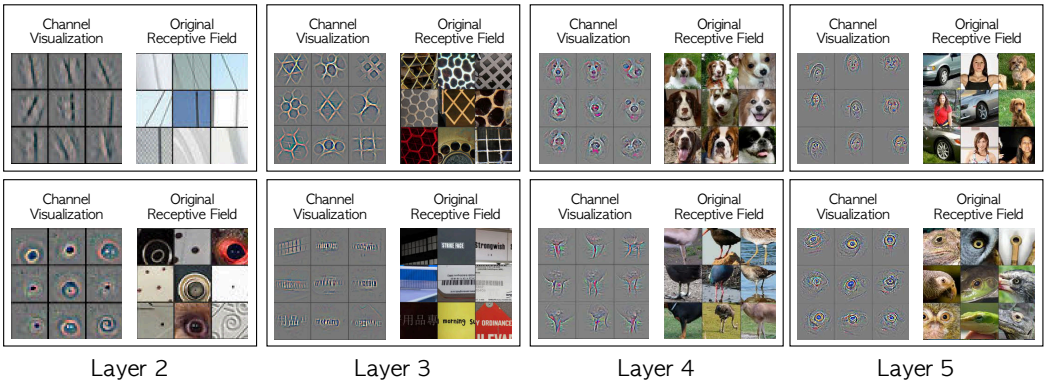



**Figure 6.12:** Visualization of the meaning of channels of a trained model.[13] We present two arbitrary channels for each of layers, 2, 3, 4, and 5 and for each channel we see the 9 images that yield the 9 highest activation values. The gray background images (channel visualization) are processed via a deconvolution network from feature space back to pixel space. The original receptive field color images are the associated receptive fields in the images that excite those activations. It can be seen that initial layers search for more elementary features and layers deeper in the network search for more refined features. Importantly, it appears like the type of features searched for in each channel are generally homogenous (although this not always the case, as is evident with the top channel presented for layer 5).

dataset (original receptive field). These channel visualizations and original receptive fields can serve as a visual interpretation of what the specific channel detects.

For example, we see that the two channels visualized in layer 2 detect simple features with one channel focusing on edges and another channel focusing on circles. As we advance deeper in the network we see that the type of visual patterns detected are much more complex. For example, the two channels presented for layer 4 detect parts of animals, and the channels of layer 5 detect such representations as well. Note however, that one of the channels in layer 5 that we present appears to detect either faces or car wheels even though these are very different objects. Hence any attempt to categorize channels based on their "meaning" alone is far from absolute. Nevertheless, a visual representation such as that in Figure 6.12 can help to understand the function of individual channels within the network.

Let us now indicate how a visualization such as that in Figure 6.12 can be created. We may focus on any arbitrary specific channel $j$ in layer $\ell$. A validation set of images is run though the network and for each image we consider the activation matrix $a_{(j)}^{[\ell]}$ of the channel. We compute $\eta_{(j)}^{[\ell]} = \max_{i,k} \left[a_{(j)}^{[\ell]}\right]_{i,k}$, where the maximum is over the pixel coordinates $(i,k) \in \{1,\dots,M_h^{[\ell]}\} \times \{1,\dots,M_v^{[\ell]}\}$. We also keep the coordinates that attain this maximum, denoted here via $i_*$ and $k_*$. The idea is to find the neuron or activation, that is maximally activated by each image in the validation set. Doing so for all images in the validation set, we then take the 9 images that achieve the maximal $\eta_{(j)}^{[\ell]}$ values and these are the 9 images used for visualization of that channel. Now for each image out of those 9 images we take the coordinate $(i_*, k_*)$ of the maximally activated neuron, and determine the receptive field of that neuron within the input image. We then present the receptive field

part of the input image for each of the 9 images. This visualization then illustrates the 9 most significant image patches for the channel at question.

As for the channel visualization part (gray background images) of Figure 6.12, a more sophisticated process is carried out on each of the 9 selected images per channel. A type of network, called a *deconvolution architecture* is constructed in parallel to the original convolutional network. This combined architecture enables transforming "feature space" back to "pixel space" for individual input images and specific neuron locations. That is, an input image to the original network is first processed. Then with a specific neuron $(i_*, k_*)$ in channel $j$ of layer $\ell$, specified, the deconvolution architecture returns an image associated with the receptive field of that particular neuron in "pixel space". While we do not specify the details of this particular deconvolution architecture, let us mention how it is used for the channel visualization. Each gray background channel visualization image in Figure 6.12, is an output in pixel space, resulting from the associated neuron, $(i_*, j_*)$ in channel $j$ of layer $\ell$ specified to the deconvolution architecture. For this all other neurons in channel $j$ of layer $\ell$ are set to 0, except for $[a_{(j)}^{[\ell]}]_{i_*,k_*}$ which is activated. The deconvolution architecture, then works backwards in the network from layer $\ell$ to layer $\ell - 1$, and back, all the way until presenting the result in pixel space. The benefit of such channel visualization images, is that they allow us to see how a single neuron $[a_{(j)}^{[\ell]}]_{i_*,k_*}$ "appears" in "pixel space". Importantly, we see that the 9 most activated neurons in the same channel, are generally of the same nature.

## 6.5 Inception, ResNets, and Other Landmark Architectures

So far we covered the key components of convolutional neural networks and considered the VGG19 model as one concrete network example. In this section we highlight other landmark architectures within the world of convolutional neural networks. Our main goal is to highlight ideas stemming from these architectures.

In general, the book avoids historical accounts as much as possible, yet in the context of network architectures, some knowledge of the historical progression might be practically useful. We thus begin with a brief historical account naming key architectures. We then focus on three architectural ideas, namely the *network within a network* (*inception network* also known as *GoogLeNet*), *residual connections* (ResNets), and efficient model scaling as in *EfficientNet*. See also the notes and references at the end of the chapter for further information.

### A Brief Historical Account

As with many ideas in deep learning, with convolutional networks one can find quite early roots. In this case, early convolutional networks include the *Neocognitron* from the late 1970's and early 1980's and *LeNet-5* worked on during the mid 1980's until the late 1990's. Both of these networks already encompass many of the ideas presented in this chapter, yet in those days computation power was lacking and ease of implementation with software was much less advanced.

The architecture that really advanced deep learning as a whole, and particularly convolutional neural networks is *AlexNet* from 2012. At the time, it was a breakthrough in image classification, achieving state-of-the-art performance on the ImageNet dataset. The archi-

tecture consists of five convolutional layers followed by three fully connected layers. It also uses two parallel computation streams allowing the network to execute parallel forward propagation and backpropagation, using two state of the art GPUs of the time. The work on AlexNet also introduced several innovations that are now commonplace, such as the use of ReLU activation functions and dropout regularization. While today, AlexNet is probably not the first off-the shelf model that one would use, it can still be cast as the first "modern convolutional neural network". From a research and applied perspective, it was the success of AlexNet that sparked the start of the deep learning era. After the introduction and success of AlexNet, hundreds (and now many thousands) of researchers, both applied and theoretical, shifted focus towards deep learning. This heavy research effort accelerated advances in the field.

Architectures that followed AlexNet include *ZFNet* in 2013, *VGGNet* (including VGG19) in 2014, *GoogLeNet* in 2014, and *ResNet* in 2015. This short sequence of advances marks the main evolution of convolutional architectures to what they are today. In more recent years, vision tasks have also been tacked by non-convolutional networks using *transformers*. For such ideas see Chapter 7, describing transformers in the context of sequence or language models, and see Chapter 8 where we highlight how ideas from different deep learning domains interplay. Nevertheless, convolutional networks remain the bread and butter of modern computer vision. A recent advance that we cover below is *EfficientNet*. This set of models tries to optimally scale models to balance performance and model size.

## Inception and Networks within a Network

The *inception network*, also called *GoogLeNet*, works by composing multiple sub-networks into a bigger network. This idea is sometimes called a *network within a network*. Each sub-network is called an *inception module* and such a module uses multiple filter sizes in parallel.

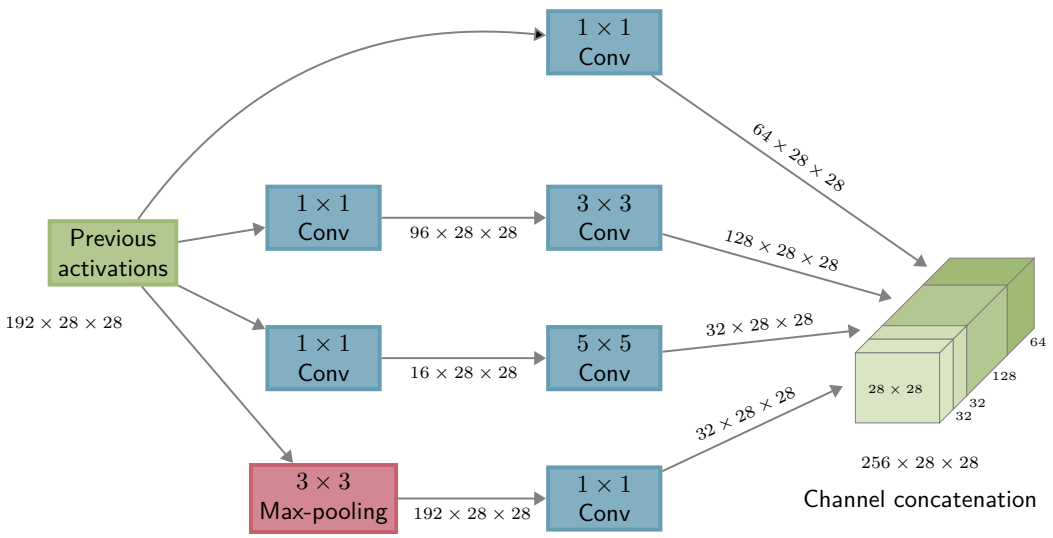



**Figure 6.13:** One form of an inception module, playing part in an inception network. The key idea is parallel computation of various paths followed by a concatenating of the outputs from all paths.

Figure 6.13 illustrates an example of one such inception module. In this example a volume of previous activations of dimension $192 \times 28 \times 28$ is transformed to an output volume of dimension $256 \times 28 \times 28$ (the number of channels grows from 192 to 256). Inside the inception module, there are four parallel paths, each operating independently and producing its own set of output channels. Then the outputs of these paths are concatenated.

The different paths of the inception module are designed to handle different scales and resolutions. The first path has $1 \times 1$ convolutions and yields 64 output channels. The second path has $1 \times 1$ convolutions followed by $3 \times 3$ convolutions. This path produces 128 output channels (with an intermediate number of 96 channels). The third path is similar, yet uses $5 \times 5$ convolutions instead of $3 \times 3$. It results in 32 output channels with 16 intermediate channels. Finally the last path starts with a max-pooling operation with a max pooling stride of 1, such that there is no reduction in spatial dimension, but rather only a non-linear operation. This is then followed by $1 \times 1$ convolutions. This path results in 192 output channels. All convolutions have ReLU activations and where needed, there is padding such that the desired spatial dimensions are respected. It should be noted that when the inception network was developed, mass experiments were conducted to seek near-optimal settings for this inception module and similar ones.

The essence of the inception network is to interconnect such inception modules in series. The concatenated channels that result as output from one module are given as input to the next module. One aspect of this interconnection is that the number of channels generally grows down the network. To mitigate such channel explosion, one by one convolutional layers are placed between some of the inception modules. Such layers have been termed *bottleneck layers*. Another aspect introduced with such networks was intermediate loss functions. Here the idea is that in addition to the final loss function at the exit of the network, the loss is also computed at various intermediate "exit points" and the gradient based optimization uses the sum of all loss functions.

Empirically, in 2014, the introduction of GoogLeNet outperformed other networks at the time and importantly these types of networks appear to strike a balance between accuracy and computational efficiency. The original GoogLeNet has about 6.8 million parameters and this is much less than the 143.7 million parameters of VGG19. GoogLeNet can be viewed as having 22 parameterized layers with a total of 9 inception modules, two convolutional layers as initial layers, and only a single dense layer at the output. Practically, these days when one wishes to use an off the shelf trained convolutional neural network, some variant of GoogLeNet is often a prime choice.

## Residual Connections

Recall early discussions in Section 2.5, and in particular Figure 2.9. There we claimed that in general, as model complexity grows we expect training error to decrease simply because our model is able to capture more complex relationships. With deep learning one would also hope to see this type of phenomena when adding layers. However, this is only partially true. Empirically it has been observed that when deep learning models get extremely deep with dozens or hundreds of layers, training error actually starts to increase. In other words, as we add more layers to a neural network, its training error initially decreases, but after a certain depth, the network's accuracy on the training set starts to saturate and sometimes degrades. One reason for this phenomenon, which is often termed a *degradation problem*, stems from vanishing and exploding gradient issues. When a gradient is backpropagated

through multiple layers, it can become extremely small, causing the weights in the earlier layers to receive almost no updates during training. As a result, the network's ability to learn and generalize is reduced. See Section 5.4 for a discussion of vanishing and exploding gradients.

Some further insight into the computational problems for very deep models is as follows. We may hypothesize that good parameters, $\theta^{[\ell]}$, for layer $\ell$, are such that the operation of the layer $f^{[\ell]}_{\theta^{[\ell]}}(\cdot)$ is approximately an identity. Namely the input to the layer, $a^{[\ell-1]}$, and the output, $a^{[\ell]}$, are ideally very similar. This can be hypothesized because with deep models we would expect individual layers to only apply minor variations on their inputs. If we accept such an hypothesis then we immediately get insight into some of the numerical and computational problems that learning entails. Specifically, learning functions close to $f^{[\ell]}_{\theta^{[\ell]}}(u) = u$ is often not trivial - for example consider a pure convolutional layer and observe that for it to be an identity function the convolution kernel requires to be all zeros except for a single entry that is 1. For this, iterating over parameters of convolutional layers until they become close to identity requires many gradient descent steps and can run into numerical problems.

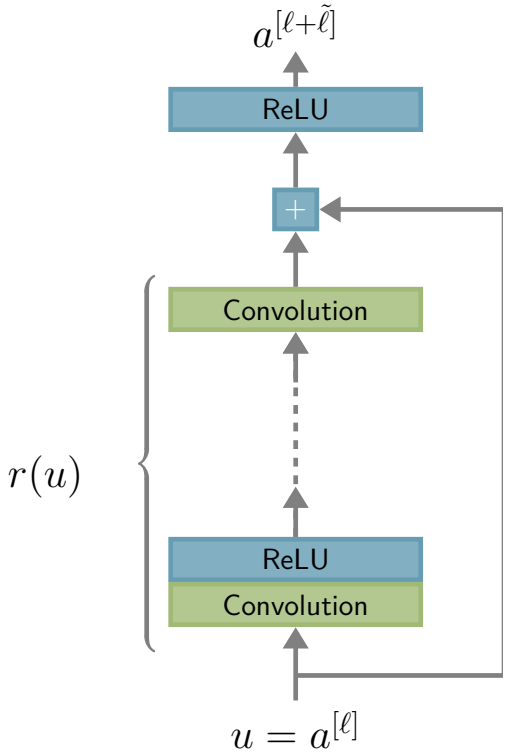



**Figure 6.14:** A shortcut connection (residual connection) as part of a residual network.

An approach to overcome this problem is to use *shortcut connections* as in Figure 6.14. Here the key idea is to take the input before a given layer (or sequence of layers), and bypass the layer (or the sequence of layers). Then the bypassed information is added to the output down the network, typically before the application of an activation function. Note that as

we bypass layers, it may be that channel dimensions are different. In such a case, we use one by one convolutions, and similarly we may use padding, stride, and pooling to adjust the spatial dimensions if needed.

Mathematically, and continuing with the hypothesis that layers should be close to identity functions, we may view the shortcut connection approach as a means to set the bypassed layer (or layers) to a function that approximately outputs zero. To see this return to Figure 6.14 and assume that $r(u) \approx 0$. This then makes the operation of the whole sequence of layers with a bypass close to the identity. Specifically in the figure we bypass $\tilde{\ell}$ layers, and if $r(u) \approx 0$, then $a^{[\ell+\tilde{\ell}]} \approx a^{[\ell]}$. Due to this reason the shortcut connections are also sometimes called *residual connections* and the whole architecture is called a *ResNet*. The usage of the term, "residual" implies that by adding a shortcut connection, we are now learning $r(u)$ to as a deviation from zero, or a residual.

When the ResNet idea was introduced in 2015, networks of depths of dozens and even more than one hundred layers were able to be efficiently trained. This elegant and simple idea allows us to learn residuals instead of actual transformations. Ideas from ResNets propagated to other aspects of deep learning beyond convolutional neural networks, such as for example some sequence models presented in the next chapter. There are also models that combine residual connections and inception modules, and these models are near the state of the art of convolutional neural networks.

### EfficientNet Models

*EfficientNet* is a family of convolutional neural network architectures that were developed with the aim of providing better accuracy and efficiency in terms of model size and computation cost. The key idea is to systematically scale up the dimensions of the network's parameters (such as depth, width, and resolution) in a balanced way, while also introducing a new compound scaling method that optimizes these dimensions based on a set of pre-defined constraints. This allows users to choose which form of EfficientNet model they want, in a way that balances the number of parameters and the performance of the model. Figure 6.15 plots the parameter count vs. performance tradeoffs of efficient net models. The models are named B0, B1, ...,B7 where B0 is the most lightweight model in terms of parameter counts, and B7 is the most computationally demanding model. It is seen that EfficientNet dominates other popular models.

## 6.6 Beyond Classification

The sections above focus on the internals of convolutional neural networks. For simplicity in those sections, we discuss the task of image classification, e.g. determining if an image is that of a cat or a dog. However, there are several other important image analysis tasks that are also handled with convolutional neural networks. These tasks deal with analysis and understanding of an image including the location of objects, the count of objects, separating between different semantic features of the image, and more. Our purpose in this section is to highlight such tasks. For this we present a brief overview of key *computer vision* developments that use convolutional neural networks for tasks beyond classification.

[14] Image thanks to M. Tan and Q. V. Le, taken from "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", [395]. See also [396].

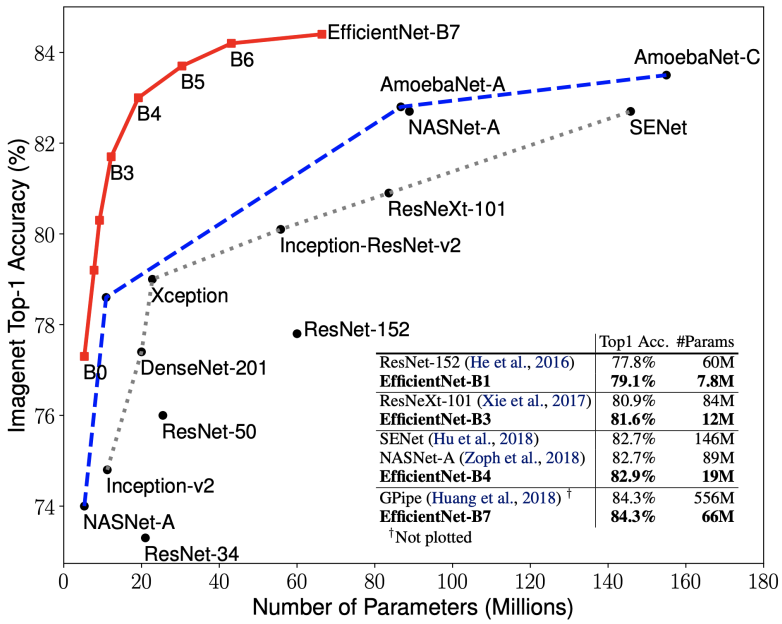



**Figure 6.15:** Performance of various convolutional models as well as efficient net.[14]

In terms of the input data, it is important to keep in mind that not all data is of the form of monochrome or color images. Within computer vision, one often deals with *image sequences* (short movies), or images that have more than 3 channels. For example, some images may also have a *distance channel* capturing the distance from the camera per pixel. Further, non-image data can also be handled via convolutional networks. One such example is *fMRI (functional magnetic resonance imaging)* data which is 4 dimensional in nature as it records the state of physical locations in three dimensions over time. Nevertheless, most of our attention in this section is restricted to images.

## Convolutional Networks and Key Computer Vision Tasks

As mentioned above, classification serves as a simple and useful example. For an input image $x$, a convolutional neural network $f_\theta(\cdot)$, has output $\hat{y} = f_\theta(x)$ which is a vector of probabilities where the highest probability typically determines the appropriate label for the image. As was evident from our detailed study of the VGG19 model in Section 6.4 and other architectures of Section 6.5, initial layers of the model $f_\theta(\cdot)$ are typically convolutional, and the final layers are typically fully connected layers. These final layers help transform the internal derived features in the network into the output vector of probabilities $\hat{y}$. When one considers tasks other than classification, it is often common to replace the final layers of the network with other layers such that the output $\hat{y}$ suites the desired task. With such a replacement we typically keep in the initial layers as is.

[15]Image (b) is thanks to J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, taken from "You only look once: Unified, real-time object detection", [346]. Image (c) is thanks to H. Lai, S. Xiao, Y. Pan, Z. Cui, J. Feng, C. Xu, J. Yin, and S. Yan, taken from "Deep recurrent regression for facial landmark detection", [246]. Image (d) is attributed to B. Palac under the creative commons license and available via Wikimedia

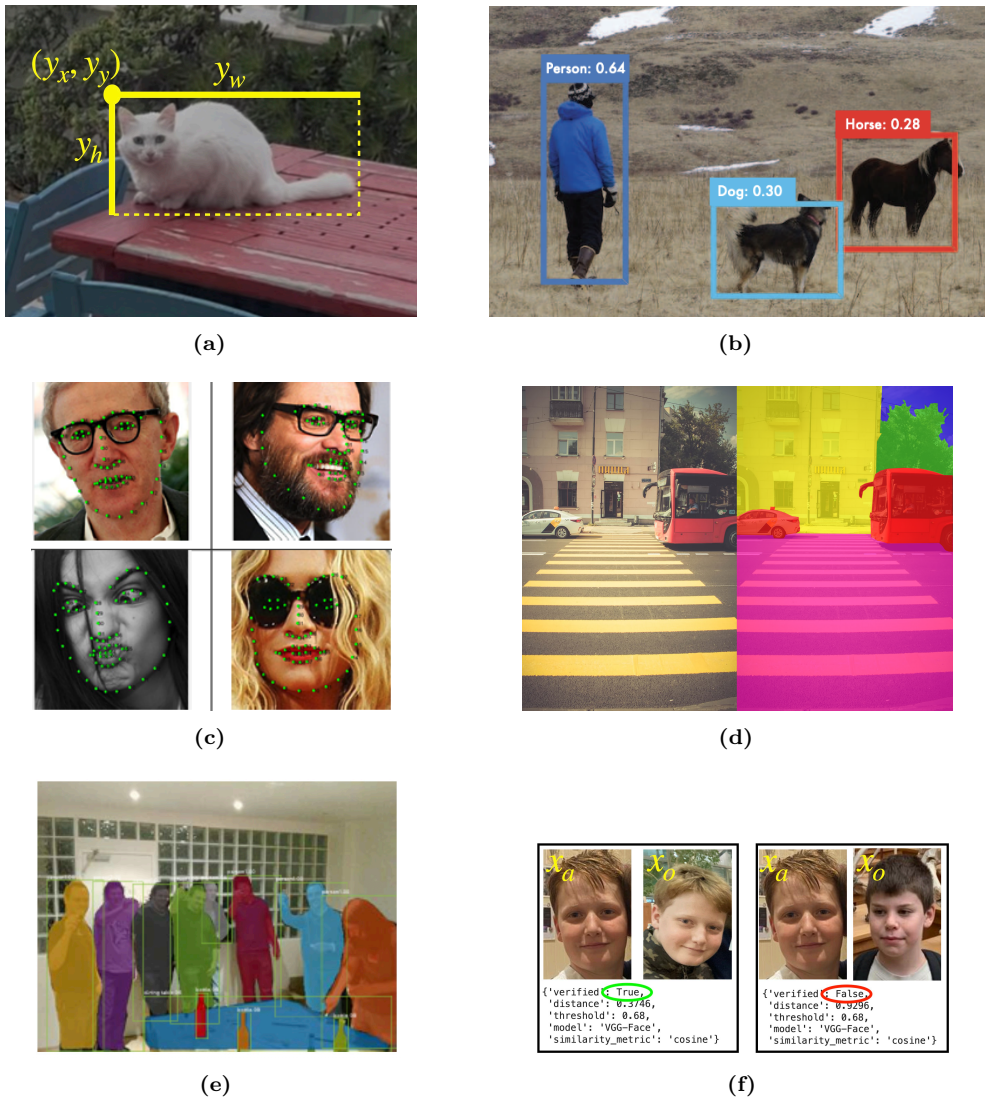



**Figure 6.16:** Illustrations[15] of some common computer vision tasks beyond classification: (a) Object localization. (b) Object detection. (c) Landmark detection. (d) Semantic segmentation. (e) Instance segmentation. (f) Identification (face recognition).

Let us now get a feel for some of these tasks and in each case consider some possible structure for the output $\hat{y}$. Figure 6.16 illustrates key computer vision tasks for images. In (a) we see *object localization* which is the task of identifying the location of an object in an image, as well as possibly the type of object in which case the task is called *localization and classification*. In (b) we see *object detection* which is the task of detecting multiple instances of an object in an image, also separating between the objects and classifying their type. In (c) we see *landmark detection* which is the task of identifying the specific pixel locations of landmarks in an image. In (d) we see *semantic segmentation* which is the process of

Commons. Image (e) is thanks to K. He, G. Gkioxari, P. Dollár, and R. Girshick, taken from "Mask R-CNN", [170].

classifying each individual pixel to be of a different class from a finite set of classes (pixel wise classification). In (e) we see *instance segmentation* which finds different instances of objects in the image and separates pixels to be of different instances. Finally, in (f) we see the task of *identification* or more specifically *face recognition* which determines if an image is that of a specific instance (or person).

Let us now consider possible forms of the output $\hat{y}$. For object localization, (a) in Figure 6.16, $\hat{y}$ needs to contain information about a *bounding box* which locates the object. This can be in the form of $(\hat{y}_x, \hat{y}_y, \hat{y}_h, \hat{y}_w)$ where $\hat{y}_x$ and $\hat{y}_y$ are the coordinates of (say) the upper left corner of the bounding box and $\hat{y}_h$, $\hat{y}_w$ are the height and width of the bounding box, respectively. This information can also be augmented with probabilities for the respective classes (types of objects) including the possibility of having no object. For object detection, (b) in the figure, a collection of multiple bounding boxes needs to be supplied. For (c), landmark detection, a list of coordinates of the locations of landmarks comprises the output. For (d) semantic segmentation, each pixel location in the input image, $x$, has an associated probability vector of classes in the output $\hat{y}$. Hence in this case, $\hat{y}$ can be represented as a tensor with width and height dimensions the same as the input image, and a depth dimension which is the number of classes in the segmentation. For (e), instance segmentation, the output is similar to that of semantic segmentation, but instead of recording probabilities of classes, the depth dimension of the output $\hat{y}$ is used for determining the specific instance of any given pixel. Finally, in the case of identification, or face recognition, as in (f) of Figure 6.16, the output is often just a probability as in a binary classifier, since the task is to determine if a face image matches a given pre-stored template or not. Note that in this case, the input $x$ is typically composed of two images, where one image, say $x_a$, is the template of the person (e.g. a stored image in a security database), and the other image, say $x_o$, is the other image.

There are many ideas that have gone into developing architectures for handling tasks (a) – (f). Some of these ideas stem from vision analysis research, prior to the era of deep learning, while other ideas evolved in parallel to deep learning in recent years. Object localization and classification as in (a) is a particularly simple example and for this we provide more details below. Similarly, identification (face recognition) is also worth consideration and we provide more details below. Landmark detection (c) is handled easily also in a similar spirit to object localization and classification; we omit the details. The other tasks including object detection (b), semantic segmentation (d), and instance segmentation (e), are each big topics of their own and we leave investigation of these for further reading. See the notes and references at the end of the chapter.

## Object Localization

To get a feel for object localization assume that we wish to train a convolutional neural network that operates on an input image $x$ and determines if the image contains a `bird` or a `plane` (classification). The model's second goal is to determine the specific location $(\hat{y}_x, \hat{y}_y, \hat{y}_h, \hat{y}_w)$ of that object (localization). Images with multiple birds or planes are not considered. Images without a bird and without a plane are possible and in this case the output yields `nothing`. One way to encode the output is $\hat{y} = (\hat{p}_{\texttt{nothing}},\ \hat{p}_{\texttt{bird}},\ \hat{p}_{\texttt{plane}}, \hat{y}_x, \hat{y}_y, \hat{y}_h, \hat{y}_w)$, where as in standard classification examples $(\hat{p}_{\texttt{nothing}},\ \hat{p}_{\texttt{bird}},\ \hat{p}_{\texttt{plane}})$ is a probability vector, and the other coordinates define a bounding box.

Here an output that has $\hat{p}_{\texttt{nothing}}$ greater than each of $\hat{p}_{\texttt{bird}}$ and $\hat{p}_{\texttt{plane}}$ implies a prediction of no bird and no plane. On the contrary if $\hat{p}_{\texttt{bird}}$ is the highest probability then the output

implies there is a bird, located in the bounding box $(\hat{y}_x, \hat{y}_y, \hat{y}_h, \hat{y}_w)$. Similarly for the other class, `plane`.

In terms of training data, for each input image we denote the output as $y$ where images without a bird or a plane are labeled as, $y = (1, 0, 0, \emptyset, \emptyset, \emptyset, \emptyset)$, where $\emptyset$ are "do not care" values. Images with a bird are labeled as $y = (0, 1, 0, y_x, y_y, y_h, y_w)$ where the bounding box $(y_x, y_y, y_h, y_w)$ is typically based on a manual determination by a human annotator. Similarly, images with a plane are labeled as $y = (0, 0, 1, y_x, y_y, y_h, y_w)$.

We now construct a loss function that captures closeness of $\hat{y}$ and $y$. For this we first separate the classification and localization objectives into a loss $C_{\text{classification}}(\theta\,;\,\hat{y}, y)$ and $C_{\text{localization}}(\theta\,;\,\hat{y}, y)$. The former depends only on the probability components in $\hat{y}$ and $y$, and the latter depends only on the bounding box components in $\hat{y}$ and $y$. For the classification loss, we use categorical cross entropy as in (3.31). For the localization loss, we use a mean squared error as in (2.12) or some variant, applied to the four bounding box components.

The two separate losses are then combined such that the loss for a specific observation is,

$$C_{\text{classification}}(\theta\,;\,\hat{y}, y) \;+\; \gamma \cdot (1 - y_1) \cdot C_{\text{localization}}(\theta\,;\,\hat{y}, y),$$

where $\gamma > 0$ is a hyper-parameter used to weigh the two losses and taken as $\gamma = 1$ by default. Observe that $y_1 = 1$ when the label is `nothing` and is otherwise 0 and thus for labels in the training data without a bird or a plane only the classification objective is used.

To perform object localization, say with a model like VGG19, the network can be modified by adding additional layers at the end of the architecture to predict the coordinates of the bounding box. This can be achieved by attaching a regression head to the output of the final convolutional layer of the network. The regression head consists of fully connected layers that predict the coordinates of the bounding box. Such simple modifications of networks that were otherwise designed for classification are always possible.

## Face Recognition, Siamese Networks, and Triplet Loss

Let us get a feel for how identification (face recognition) as in Figure 6.16 (f) can be implemented both in production and training. First let us consider the simplified use of such a task. Say a face identification system needs to be able to recognize faces where in production one may have an *anchor* face image $x_a$ stored. With each use, the anchor needs to be compared to another image $x_o$. For example every "login" is based on a new $x_o$ image and the system needs to determine if $x_o$ is of the same person as $x_a$ or not. In contrast to other tasks discussed in the book, here we do not have the ability to train a network for a particular person (or face), and similarly we do not have many different face images of the same person. Hence this setup requires a slightly different architecture.

One type of architecture useful for this task is a *siamese network*, illustrated schematically in Figure 6.17. The idea is that two parallel replicas of a convolutional neural network $f_\theta(\cdot)$ are used, one applied on $x_a$ and the other on $x_o$. The output of each of these networks is an embedding vector. Now since we have two embedding vectors, $f_\theta(x_a)$ and $f_\theta(x_o)$, we can compare them and see if they are likely associated with face images of the same person or not. One approach for this comparison is as in Figure 6.17 using a *comparison network*, $f^c_{\theta_c}(\cdot, \cdot)$ for binary classification (output is a probability) with parameters $\theta_c$. Hence in production we can determine, `same` if the output probability $f^c_{\theta_c}(f_\theta(x_a), f_\theta(x_a))$ is greater than a threshold,

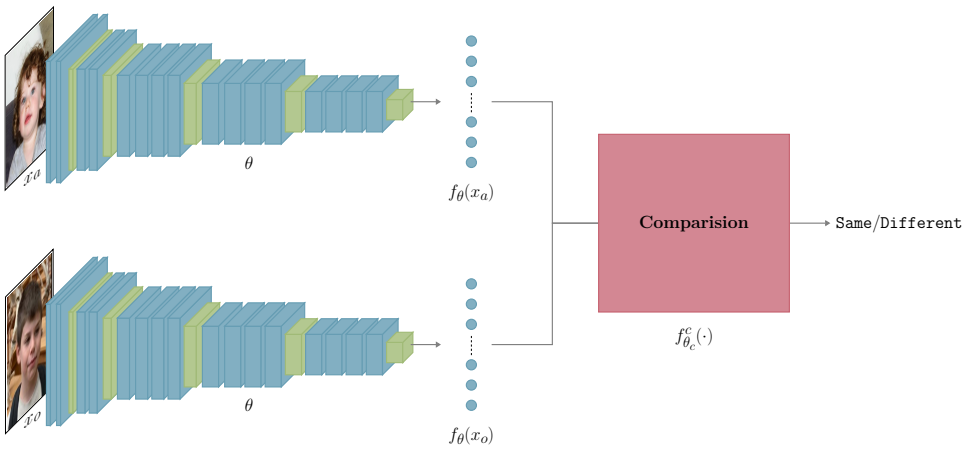



**Figure 6.17:** A schematic of a siamese network architecture for identification (face recognition). The two parallel convolutional neural networks both share the same parameters $\theta$, and one operates on $x_a$ while the other operates on $x_o$. The outputs of these networks are embedding vectors. These are then compared via a comparison module which may be a neural network with parameters $\theta_c$ and has output indicating if `same` or `different`.

or otherwise determine `different`. The comparison network is not too complex, and is often a shallow logistic regression model or similar. With such an architecture, the learned parameters $\theta$ and $\theta_c$ are not designed for one particular face, but rather for any possible face.

Let us describe a simplified approach for training such a model. We can first treat the parameters $\theta$ as known, say from a pertained or fine tuned model, and focus on learning the parameters of the (smaller) comparison network $\theta_c$. For this, the training data can be of the form, $\mathcal{D} = \big\{(x_a^{(1)}, x_o^{(1)}, y^{(1)}), \ldots, (x_a^{(n)}, x_o^{(n)}, y^{(n)})\big\}$, where each tuple $(x_a^{(i)}, x_o^{(i)}, y^{(i)})$, has an anchor image $x_a^{(i)}$, another image $x_o^{(i)}$, and a binary label $y^{(i)} \in \{0, 1\}$ with the value based on the images being `different` (0) or `same` (1). With such a dataset all that is required is to train the binary classifier $f^c_{\theta_c}(\cdot)$.

A related useful concept for training siamese networks is the *triplet loss*. Say for simplicity that now our goal is to learn $\theta$ for $f_\theta(\cdot)$, ignoring the comparison network. For this we can setup a slightly different dataset of the form $\mathcal{D}_{\text{triplet}} = \big\{(x_a^{(1)}, x_d^{(1)}, x_s^{(1)}), \ldots, (x_a^{(n)}, x_d^{(n)}, x_s^{(n)})\big\}$ where now each $(x_a^{(i)}, x_d^{(i)}, x_s^{(i)})$ has an anchor face image $x_a^{(i)}$ as before, and also has two additional images with $x_d^{(i)}$ being a face image of a different person, and $x_s^{(i)}$ being a face image of the same person (not the exact same image as $x_a^{(i)}$). Now by applying $f_\theta(\cdot)$ on each element of this dataset, we can construct a loss function for observation $i$ as,

$$C_i(\theta\,;\, x_a^{(i)}, x_s^{(i)}, x_d^{(i)}) = \max\big\{ \underbrace{\|f_\theta(x_a^{(i)}) - f_\theta(x_s^{(i)})\|^2}_{d_{\text{same}}} - \underbrace{\|f_\theta(x_a^{(i)}) - f_\theta(x_d^{(i)})\|^2}_{d_{\text{different}}} + \alpha, 0\big\}, \quad (6.27)$$

where $\alpha > 0$ is some hyper-parameter called the *margin* and the Euclidean norm $\|\cdot\|$ can in principle be replaced by a different distance metric as well.

Let us understand the motivation behind the triplet loss (6.27). Our desire is that the embedding associated with the anchor image $x_a^{(i)}$ and embedding associated with the image of the same person $x_s^{(i)}$ be close to each other and hence $d_{\text{same}}$ should ideally be small. Similarly we wish to have the embedding of $x_a^{(i)}$ and the embedding $x_d^{(i)}$ to be distant from each other and this motivates the negative sign in front of the $d_{\text{different}}$ term which we ideally want to be large.

Now in general, when we have such an optimization with two competing criteria, $d_{\text{same}}$ which we want to be small, and $d_{\text{different}}$ which we want to be large, one approach to capture such a desire via a loss, is by pre-determining a margin $\alpha$ and considering cases where,

$$d_{\text{same}} - d_{\text{different}} \leq -\alpha, \tag{6.28}$$

as being "admissible" and otherwise "inadmissible". We can then assign a loss of 0 to admissible cases, and assign a loss that depends on $\theta$ for the inadmissible cases. This is achieved with the $\max\{\cdot, 0\}$ operation since if (6.28) is satisfied, the loss in (6.27) is 0. In contrast, when (6.28) is not satisfied (inadmissible), the loss in (6.27) is $d_{\text{same}} - d_{\text{different}} + \alpha$. Hence when using gradient descent based learning of $\theta$ for minimization of $\sum_{i=1}^{n} C_i(\theta\,;\mathcal{D}_{\text{triplet}})$, at any iteration, we drive loss down for the inadmissible observations.

The triplet loss with a properly curated dataset $\mathcal{D}_{\text{triplet}}$ has been effectively used for state of the art face recognition training. We note that when curating this dataset it is often important to preprocess the images so that $x_a^{(i)}$ and $x_d^{(i)}$ are not acutely different. We also note that with the use of the triplet loss we can add a comparison network $f_{\theta_c}^{c}(\cdot)$ which is trained as a binary classifier, after training $\theta$ with the triplet loss. In other cases, using the cosine distance between the two embedding vectors $f_\theta(x_a)$ and $f_\theta(x_o)$ suffices in production.

## Notes and References

Before we outline notes and references associated with explicit details of this chapter, here is a brief description of early *convolutional neural network* developments. Initial ideas originated in the 1950's and 1960's with the study of the visual cortices of animals, primarily by Hubel and Wiesel over a series of publications including [195] and [196]. Early concrete models that have some similarity with modern convolutional neural networks are the 1980 *neocognitron* [127] for pattern recognition, as well as the 1988 *time delay neural network* [247] for speech recognition. In the 1990s convolutional neural networks saw industrial applications for the first time with [154] for handwritten character recognition and [65] for signature verification. Other significant early works include [250] for written digit recognition, [402] for face recognition, and [412] for phoneme recognition. Finally we mention that the *LeNet-5* model developed in the late 1990's by Yann LeCun et al. [252] is recognized as an early form of contemporary convolutional neural networks and it was used for classifying $28 \times 28$ size images of grayscale handwritten digits. We also mention that in 1989 with [250] and [251], LeCun et al. developed the first multi-layered convolutional networks for handwritten character recognition trained using *backpropagation*.

The structure of convolutional layers in neural networks as we present in this chapter solidified at around the 2012–2016 period and best fits the *VGG model* [380]. This model followed the pivotal *AlexNet* model [239] from 2012 which was specifically designed for training on two parallel GPUs. Other notable convolutional architectures of this period are the *GoogLeNet* or *inception network* model of [394], the *batch normalization inception* model [203] which uses batch normalization of layer inputs, and *ResNets* which were introduced in [172]. All of these models competed in the *ImageNet challenges* of that era with the results from each model effectively outperforming those that came prior to it. Other developments included the *SqueezeNet* model of [199], which marked a key milestone in reducing parameter size and memory footprint of convolutional network without compromising accuracy; this model achieved the AlexNet-level accuracy with much fewer parameters and a much smaller memory footprint. Also, see the *Network-in-Network* model of [260] which inspired the inception networks and [388] that uses dropout mechanism to reduce overfitting on convolutional layers. See [344] for a comprehensive survey of convolutional neural networks of that time as well as the more recent survey [258]. In times closer to the publication of this book, paradigms such as *EfficientNet* appeared in [395], see also the more recent version, EfficientNet v2 in [396].

Ideas of dilation in convolutional networks were introduced in [435] for dense prediction, where the goal is to compute a label for each pixel in the image. Furthermore, dilation for residual networks is introduced in [436]. See also the discussion of *group normalization* in [426].

A general overview of *linear time invariant systems* can be found in standard texts such as [244] which is also useful for understanding basic filtering. The book [14] can provide a more mathematically rigorous foundation and can also be useful for understanding the delta function in continuous time. The probabilistic interpretation of a convolution is standard and can be found in any elementary probability textbook such as [355]. The multiplication of polynomials interpretation, also coupled with the study of the *fast Fourier transform* can be found in [93]. A simple explanation of the representation of discrete convolutions in terms of *Toeplitz matrices* can be found in [56]. For analysis of convolutions of classic image processing applications as well as many other classic image processing techniques see [207]. *The Sobel filter* is one of many convolution based filtering operations. It was developed by Sobel and Feldman, and presented at a 1968 scientific talk; see [383] for an historical review.

The rise of convolutional neural networks drove the development of many paradigms using these networks for different tasks. In terms of object detection, early works are [135] and [136] and recent work in this direction is [415] where YOLOv7 model enhances the landmark *YOLO* (you only look once) work of [346]. A recent survey on object detection can be found in [454]. The important area of semantic segmentation has received much attention with notable papers being [352] (U-net), as well as [312]. Instance segmentation is studied in [45], [170], and [267]. For additional recent surveys of the subsequent developments in semantic and instance segmentation see [405] and [290]. See also [129] for a survey of video semantic segmentation. Influential work on identification (*face recognition*) is in [368] and early ideas of siamese networks are from [84]; see also [192] and [425].

Over the years, many effective network visualization methods were developed for understanding inner layers and derived features. Before the era of great popularity of convolutional networks, the

work in [119] introduced a technique aimed at optimizing the input to maximize the activity of hidden neurons in a deep neural network. For convolutional neural networks, the *deconvolution architecture* in [441], based on previous work in [442], was a significant as it was the first work where effective visualization of internal layers was made possible. Other related important papers in this direction are [278] which introduced a technique called *network inversion* and [23] which introduced a framework called *network dissection*. A general useful survey on visual interpretability for deep learning is [447]. Other related ideas that we do not discuss in this book include deep dreaming[16] and directly using convolutional networks for neural style transfer, initially introduced in [131], and further developments reported in [113]. See also the related generative models of Chapter 8.

In terms of real world applications, these days convolutional neural networks are used in many scenarios. For *image classification* applications of convolutional neural networks, see for example [372] dealing with traffic sign recognition, and [256] for medical image classification, among many others. For a review of advances in image classification, refer to [76]. The most basic application of convolutional neural networks is with 3-dimensional tensors as appropriate for color images, yet there are other cases as well. In [450] 4-dimensional fMRI data is studied. Also videos are analyzed in [221] by treating the entire video as a bag of short clips. In particular, see [122], [379], and [416] for video-based action recognition; a brief summary of such methods is listed in [434]. In general, techniques for analyzing video data vary depending on the task at hand; see [373] for a brief survey of such tasks and the corresponding methods. We also mention that *transformer models*, as introduced in Chapter 7 have been applied to images and managed to surpass the performance of convolutional networks in certain cases when trained with huge datasets. See [227] for a survey as well as the notes and references at the end of Chapter 7.

[16] This blog post is credited for introducing the concept of deep dreaming: https://blog.research.google/2015/06/inceptionism-going-deeper-into-neural.html.