Studies in Big Data 57

M. Arif Wani
Farooq Ahmad Bhat
Saduf Afzal
Asif Iqbal Khan

Advances in Deep Learning

Springer

Studies in Big Data

Volume 57

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

M. Arif Wani · Farooq Ahmad Bhat ·
Saduf Afzal · Asif Iqbal Khan

Advances in Deep Learning

Springer

M. Arif Wani
Department of Computer Sciences
University of Kashmir
Srinagar, Jammu and Kashmir, India

Farooq Ahmad Bhat
Education Department
Government of Jammu and Kashmir
Kashmir, Jammu and Kashmir, India

Saduf Afzal
Islamic University of Science
and Technology
Kashmir, Jammu and Kashmir, India

Asif Iqbal Khan
Department of Computer Sciences
University of Kashmir
Srinagar, Jammu and Kashmir, India

ISSN 2197-6503 ISSN 2197-6511 (electronic)
Studies in Big Data
ISBN 978-981-13-6793-9 ISBN 978-981-13-6794-6 (eBook)
https://doi.org/10.1007/978-981-13-6794-6

Library of Congress Control Number: 2019932671

Preface

This book discusses the state-of-the-art deep learning models used by researchers recently. Various deep architectures and their components are discussed in detail. Algorithms that are used to train deep architectures with fast convergence rate are illustrated with applications. Various fine-tuning algorithms are discussed for optimizing the deep models. These deep architectures not only are capable of learning complex tasks but can even outperform humans in some dedicated applications.

Despite the remarkable advances in this area, training deep architectures with a huge number of hyper-parameters is an intricate and ill-posed optimization problem. Various challenges are outlined at the end of each chapter. Another issue with deep architectures is that learning becomes computationally intensive when large volumes of data are used for training. The book describes a transfer learning approach for faster training of deep models. The use of this approach is demonstrated in fingerprint datasets.

The book is organized into eight chapters:

Chapter 1 starts with an introduction to machine learning followed by fundamental limitations of traditional machine learning methods. It introduces deep networks and then briefly discusses why to use deep learning and how deep learning works.

Chapter 2 of the book is dedicated to one of the most successful deep learning techniques known as convolutional neural networks (CNNs). The purpose of this chapter is to give its readers an in-depth but easy and uncomplicated explanation of various components of convolutional neural network architectures.

Chapter 3 discusses the training and learning process of deep networks. The aim of this chapter is to provide a simple and intuitive explanation of the backpropagation algorithm for a deep learning network. The training process has been explained step by step with easy and straightforward explanations.

Chapter 4 focuses on various deep learning architectures that are based on CNN. It introduces a reader to block diagrams of these architectures. It discusses how deep learning architectures have evolved while addressing the limitations of previous deep learning networks.

Chapter 5 presents various unsupervised deep learning architectures. The basics of architectures and associated algorithms falling under the unsupervised category are outlined.

Chapter 6 discusses the application of supervised deep learning architecture for face recognition problem. A comparison of the performance of supervised deep learning architecture with traditional face recognition methods is provided in this chapter.

Chapter 7 focuses on the application of convolutional neural networks (CNNs) for fingerprint recognition. This chapter extensively explains automatic fingerprint recognition with complete details of the CNN architecture and methods used to optimize and enhance the performance. In addition, a comparative analysis of deep learning and non-deep learning methods is presented to show the performance difference.

Chapter 8 explains how to apply the unsupervised deep networks to handwritten digit classification problem. It explains how to build a deep learning model in two steps, where unsupervised training is performed during the first step and supervised fine-tuning is carried out during the second step.

Srinagar, India

M. Arif Wani
Farooq Ahmad Bhat
Saduf Afzal
Asif Iqbal Khan

Contents

About the Authors

Prof. M. Arif Wani completed his M.Tech. in Computer Technology at the Indian Institute of Technology, Delhi and his Ph.D. in Computer Vision at Cardiff University, UK. Currently, he is a Professor at the University of Kashmir, having previously served as a Professor at California State University Bakersfield. His main research interests are in gene expression datasets, face recognition techniques/algorithms, artificial neural networks and deep architectures. He has published many papers in reputed journals and conferences in these areas. He was honored with The International Technology Institute Award in 2002 by the International Technology Institute, California, USA. He is a member of many academic and professional bodies, e.g. the Indian Society for Technical Education, Computer Society of India, IEEE USA and Optical Society of America.

Dr. Farooq Ahmad Bhat completed his M.Phil. and Ph.D. in Computer Science at the University of Kashmir. His dissertation focused on ‘Efficient and robust convolutional neural network based models for face recognition’. Currently, his main interests are in artificial intelligence, machine learning and deep learning, areas in which he has published many articles.

Dr. Saduf Afzal teaches at the Islamic University of Science and Technology, Kashmir, India. She completed her BCA, MCA, M.Phil. and Ph.D. at the Department of Computer Science, University of Kashmir. She has also worked as an academic counselor for the MCA program at IGNOU University. Her main research interests are in machine learning, deep learning and neural networks. She has published many articles in high impact journals and conference proceedings.

Dr. Asif Iqbal Khan currently works as a Lecturer in the Higher Education Department, Kashmir, India. He completed his MCA, M.Phil. and Ph.D. at the Department of Computer Science, University of Kashmir. His main research interests are in machine learning, deep learning, and image processing. He is actively publishing in these areas.

Abbreviations

AE	Autoencoder
AI	Artificial intelligence
ANN	Artificial neural network
BN	Batch normalization
BP	Backpropagation
BPAG	Dropout-backpropagation with adaptive gain
BPGP	Dropout-backpropagation with pattern-based gain
CAE	Contractive autoencoder
CD	Contrastive divergence
CDBNs	Convolutional deep belief networks
CL	Convolutional layer
CNN	Convolutional neural network
CNN-AFC	CNN Architecture for Fingerprint Classification
ConvNet	Convolutional neural network
DAE	Denoising autoencoder
DBNs	Deep belief networks
DCT	Discrete cosine transform
DenseNet	Dense convolutional network
EBGM	Elastic bunch graph matching
FDR	False detection rate
GANs	Generative adversarial networks
GD	Gradient descent
GPUs	Graphics processing units
GWT	Gabor wavelet transform
ICA	Independent component analysis
IIIT-D	Indraprastha Institute of Information Technology, Delhi
ILSVRC	ImageNet Large-Scale Visual Recognition Challenge
ILSVRV	ImageNet Large-Scale Visual Recognition Competition
KL	Kullback–Leibler
LDA	Linear discriminant analysis

LRN	Local response normalization
M-DBNs	Modular deep belief networks
MDR	Missed detection rate
MLP	Multilayer perceptron
MrDBN	Multiresolution deep belief network
MSE	Mean squared error
NIST	National Institute of Standards and Technology
NIST-DB4	NIST Special Database 4
ORL	Olivetti Research Ltd face dataset
PCA	Principal component analysis
RBF	Radial basis function
RBM	Restricted Boltzmann machine
ReLU	Rectified Linear Unit
RMS	Root mean square
RoBMs	Robust restricted Boltzmann machines
RTRBMs	Recurrent temporal restricted Boltzmann machines
SGD	Stochastic gradient descent
SVM	Support vector machine
TRBM	Temperature-based restricted Boltzmann machine

Chapter 1
Introduction to Deep Learning

1.1 Introduction

Machine learning systems, with shallow or deep architectures, have ability to learn and improve with experience. The process of machine learning begins with the raw data which is used for extracting useful information that helps in decision-making. The primary aim is to allow a machine to learn useful information just like humans do. At abstract level, machine learning can be carried out using following approaches:

Supervised learning adapts a system such that for a given input data it produces a target output. The learning data is made up of tuples (*attributes*, *label*) where "attributes" represent the input data and "label" represents the target output. The goal here is to adapt the system so that for a new input the system can predict the target output. Supervised learning can use both continuous and discrete types of input data.

Unsupervised learning involves data that comprises of input vectors without any target output. There are different objectives in unsupervised learning, such as clustering, density estimation, and visualization. The goal of clustering is to discover groups of similar data items on the basis of measured or perceived similarities between the data items. The purpose of density estimation is to determine the distribution of the data within the input space. In visualization, the data is projected down from a high-dimensional space to two or three dimensions to view the similar data items.

Semi-supervised learning first uses unlabeled data to learn a feature representation of the input data and then uses the learned feature representation to solve the supervised task. The training dataset can be divided into two parts: the data samples with corresponding labels and the data samples where the labels are not known. Semi-supervised learning can involve not providing with an explicit form of error at each time but only a generalized reinforcement is received giving indication of how the system should change its behavior, and this is sometimes referred to as reinforcement

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_1

learning. Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control and efficient webpage indexing.

1.2 Shallow Learning

Shallow architectures are well understood and perform good on many common machine learning problems, and they are still used in a vast majority of today's machine learning applications. However, there has been an increased interest in deep architectures recently, in the hope to find means to solve more complex real-world problems (e.g., image analysis or natural language understanding) for which shallow architectures are unable to learn models adequately.

1.3 Deep Learning

Deep learning is a new area of machine learning which has gained popularity in recent past. Deep learning refers to the architectures which contain multiple hidden layers (deep networks) to learn different features with multiple levels of abstraction. Deep learning algorithms seek to exploit the unknown structure in the input distribution in order to discover good representations, often at multiple levels, with higher level learned features defined in terms of lower level features.

Conventional machine learning techniques are restricted in the way they process the natural data in its raw form. For decades, constructing a pattern recognition or machine learning system required considerable domain expertise and careful hand engineering to come up with a feature extractor that transformed the raw data (such as pixel values of an image) into suitable internal representation or feature vector from which the learning system, such as a classifier, could detect or classify patterns in the input. Deep learning allows inputting the raw data (pixels in case of image data) to the learning algorithm without first extracting features or defining a feature vector. Deep learning algorithms can learn the right set of features, and it does this in a much better way than extracting these features using hand-coding. Instead of handcrafting a set of rules and algorithms to extract features from raw data, deep learning involves learning these features automatically during the training process.

In deep learning, a problem is realized in terms of hierarchy of concepts, with each concept built on the top of the others. The lower layers of the model encode some basic representation of the problem, whereas higher level layers build upon these lower layers to form more complex concepts.

Given an image, the pixel intensity values are fed as inputs to the deep learning system. A number of hidden layers then extract features from the input image. These hidden layers are built upon each other in a hierarchal fashion. At first, the lower level layers of the network detect only edge-like regions. These edge regions are then used to define corners (where edges intersect) and contours (outlines of objects). The layers in the higher level combine corners and contours to lead to more abstract "object parts" in the next layer. The key aspect of deep learning is that these layers of features are not handcrafted and designed by human engineers; rather, they are learnt from data gradually using a general-purpose learning procedure.

Finally, the output layer classifies the image and obtains the output class label—the output obtained at the output layer is directly influenced by every other node available in the network. This process can be viewed as hierarchical learning as each layer in the network uses the output of previous layers as "building blocks" to construct increasingly more complex concepts at the higher layers. Figure 1.1 compares traditional machine learning approach based on handcrafted features to deep learning approach based on hierarchical representation learning.

Specifically, in deep learning meaningful representations from the input data are learnt by putting emphasis on buiding complicated mapping using a series of sim-

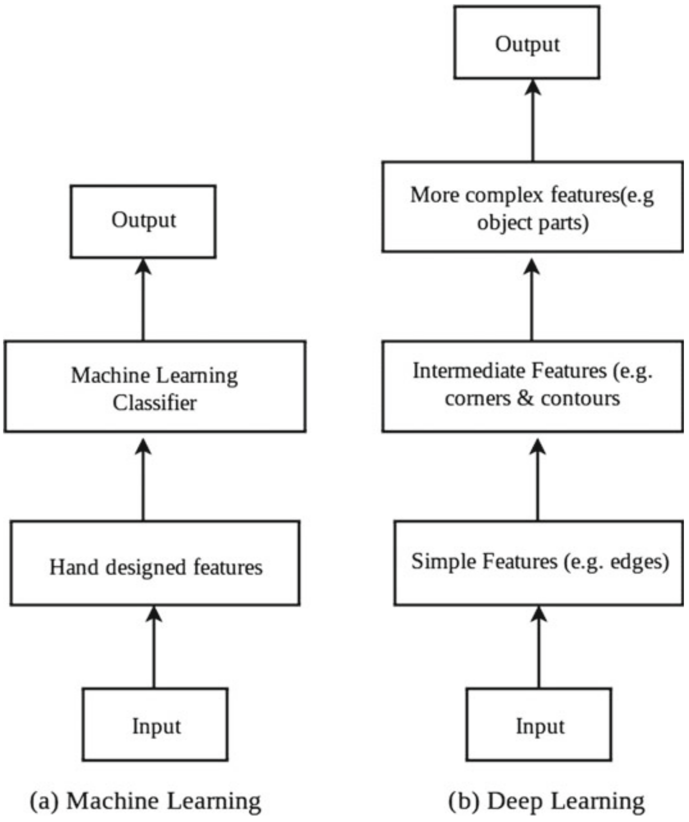

Output
Machine Learning Classifier
Hand designed features
Input
(a) Machine Learning
Output
More complex features(e.g object parts)
Intermediate Features (e.g. corners & contours
Simple Features (e.g. edges)
Input
(b) Deep Learning

Fig. 1.1 **a** Conventional machine learning using hand-designed feature extraction algorithms **b** deep learning approach using hierarchy of representations that are learnt automatically ple mappings. The word “deep” refers to learning successive layers of increasingly meaningful representations of input data. The number of layers used to model the data determines the depth of the model. Current deep learning often involves learning tens or even hundreds of successive layers of representation from the training data automatically. The conventional approaches to machine learning often focus on learning only one or two layers of representations of data; such approaches are often categorized as shallow learning. Deep learning and machine learning are subfields of Artificial Intelligence (AI). Figure 1.2 illustrates the relationship between AI, machine learning, and deep learning.

In deep learning, the successive layers of representations may be learned via submodels, which are structured in the form of layers stacked on the top of each other. As deep learning network has typically more layers and parameters, it has the potential to represent more complex inputs. Although deep learning has been around since 1980s, it was relatively unpopular for several years as the computational infrastructure (both hardware and software) was not adequate and the available datasets were quite small. With the decline in the popularity of the conventional neural networks, it was only recently that deep networks made a big reappearance by achieving spectacular results in speech recognition and computer vision tasks. Some of the aspects that helped in the evolution of deep networks are listed below:

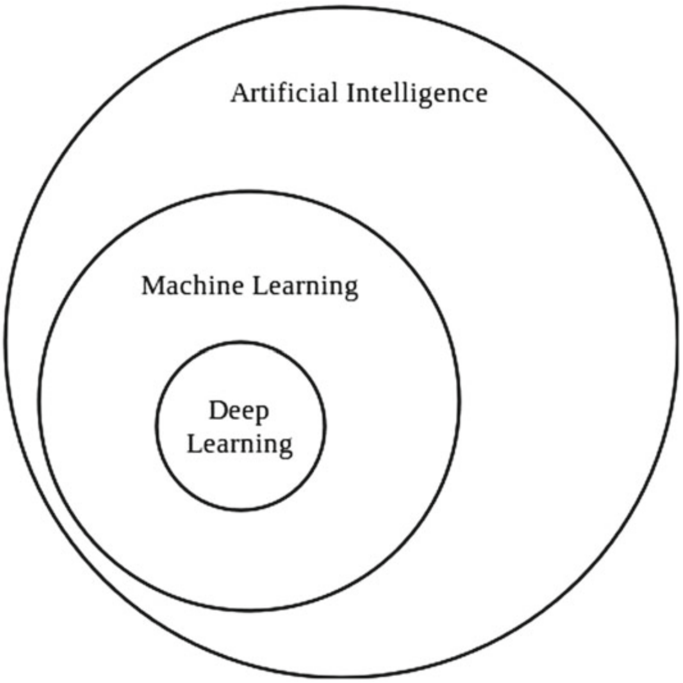


Fig. 1.2 Relationship between AI, machine learning, and deep learning

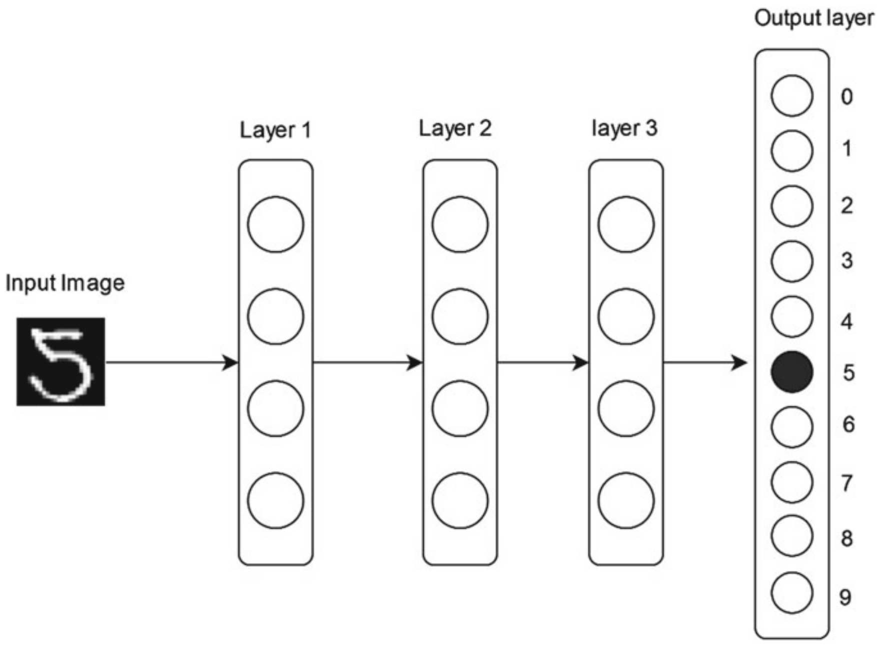


Fig. 1.3 A deep learning network for digit classification

- Improved computational resources for processing massive amounts of data and training much larger models.
- Automatic feature extraction.

The term artificial neural networks has a reference to neuroscience but deep learning networks are not models of the brain; however, deep learning models are formulated by only drawing inspiration from the understanding of biological brain. Not all the components of deep models are inspired by neuroscience; some of them come from empirical exploration, theory, and intuition. The neural activity in our brains is far more complex than might be suggested by simply studying artificial neurons. The learning mechanisms used by deep learning models are in no way comparable to the human brain, but can be described as a mathematical framework for learning representations from data.

Figure 1.3 shows an example of a deep learning architecture that can be used for character recognition. Figure 1.4 shows representations that are learned by the deep learning network. The deep network uses several layers to transform the input image (here a digit) in order to recognize what the digit is. Each layer performs some transformations on the input that it receives from the previous layers.

The deep network transforms the digit image into representations that tend to capture a higher level of abstraction. Each hidden layer transforms the input image into a representation that is increasingly different from the original image and increasingly informative about the final result. The representations learnt help to distinguish

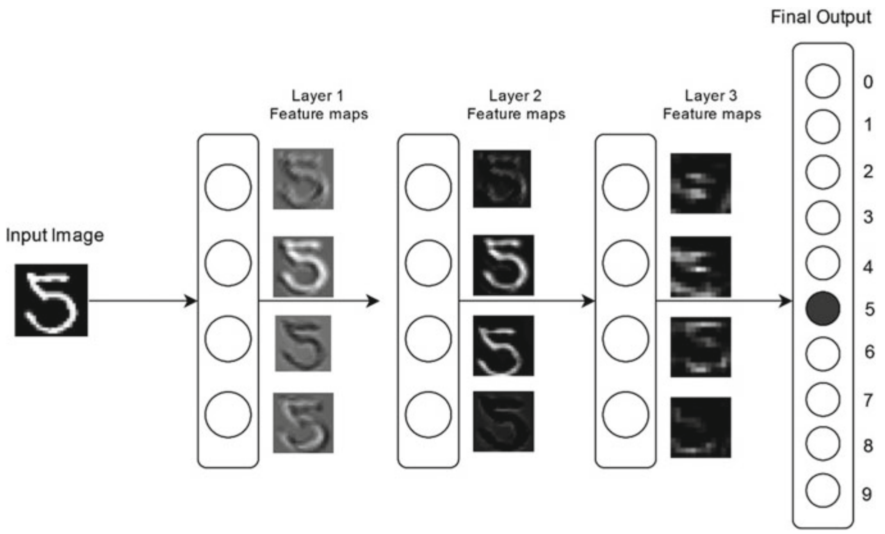


Fig. 1.4 Representations learnt by a deep network for digit classification during the first pass. Network structural changes can be incorporated that result in desired representations at various layers

between different concepts which in turn help to find out similarities between it. Deep network can be thought of as a multistage distillation information operation, where layers use multiple filters on the information to obtain an increasingly transformed form of information (i.e., the information useful with regard to some task).

In summary, a deep learning network constructs features at multiple levels, with higher features constructed as functions of lower ones. It is a fast-growing field that circumvents the problem of feature extraction which is used as a prelude by conventional machine learning approaches. Deep learning is capable of learning the appropriate features by itself, requiring little steering by the user.

1.4 Why to Use Deep Learning

The choice of features that represent a given dataset has a profound impact on the success of a machine learning system. Better results cannot be achieved without identifying which aspects of the problem need to be included for feature extraction that would be more useful to the machine learning algorithm. This requires a machine learning expert to collaborate with the domain expert in order to obtain a useful feature set. A biological brain can easily determine which aspects of the problem it needs to focus on with comparatively little guidance. This is not the case with the artificial agents, thereby making it difficult to create computer learning systems that can respond to high-dimensional input and perform hard AI tasks.

Machine learning practitioners have spent a huge time to extract informative features from the data. At the time of Big Bang introduction of deep learning, the state-of-the-art machine learning algorithms had already took decades of human effort to accumulate relevant set of features required to classify the input.

Deep learning has surpassed those conventional algorithms in accuracy as the features are learnt from the data using a general-purpose learning procedure instead of being designed by human engineers. Deep networks have demonstrated dramatic improvements in computer vision and have dramatically improved machine translation, and have taken off as an effective AI technique that has the ability to recognize spoken words nearly as good as a person can. It has achieved not only the excellent accuracy in machine learning modeling, but it has also demonstrated outstanding generalization power that has even attracted scientists from other academic disciplines. It is now being used as a guide to make key decisions in fields like medicine, finance, manufacturing, and beyond.

Deep learning grew to prominence in 2007, with promising results on perceptual problems such as hearing and seeing problems that humans are very good at, but have long been subtle for the machines. It has enabled the computer scientists to harness the vast computational power and use large volumes of data—audio, video, to teach computers how to do things that seem natural and intuitive for humans, such as spotting objects in the photos, recognizing words or sentences, and translating a document into other language. It has made it possible for machines to output the transcript from an audio clip–speech recognition, to identify whether a mail is spam or not, likelihood of whether a customer will repay his loan and so on; as long as there is enough data to train machines, the possibilities are endless. It has achieved state-of-the-art results on many applications, such as natural language parsing, language modeling, image and character recognition, playing the challenging game of Go, pixels-to-controls video game playing, and in other applications.

Today, many tech giant companies—Facebook, Baidu, Amazon, Microsoft, and Google—have commercially deployed deep learning applications. These companies have vast amount of data and deep learning works well whenever there are vast volumes of data and complex problems to solve. Many companies are using deep learning to develop more helpful and realistic customer service representatives—Chatbots.

In particular, deep learning has made good impact in historically difficult areas of machine learning:

- Near-human-level image classification;
- Near-human-level speech recognition;
- Near-human-level handwriting transcription;
- Improved self-driving cars;
- Digital assistants such as Google Now, Microsoft Cortana, Apple's Siri, and Amazon Alexa;
- Improved ad targeting, as used by Google, Baidu, and Bing;
- Improved search results on the web;
- Ability to answer natural language questions; and
- Superhuman Go, Shogi, and Chess playing.

The exceptional performance of deep models can be mainly attributed to their flexibility in representing a rich set of highly nonlinear functions as well as the devised methods for efficient training of these powerful networks. Furthermore, employing various regularization techniques ensured that deep models with huge numbers of free parameters are statistically desirable in the sense that they will generalize well to unseen data. The automatic and generic approach of feature learning in deep models enables one to use them across different applications (e.g., image classification, speech recognition, language modeling, and information retrieval) with relatively little adjustments. Therefore, deep models seem to be domain-oblivious in the sense that in order to use it across different applications, only a small amount of domain-specific customizations is required. Ideally, the domain-obliviousness of deep networks is advantageous, as having access to a universal and generic model reduces the hassles of adapting for new applications.

Deep learning is still in its infancy, but it is likely that deep learning will have many successes in the near future as it requires little hand engineering and thus can take advantage of vast amount of data and computation power. Deep learning has succeeded in previously unsolved problems which were quite difficult to resolve using machine learning as well as other shallow networks. The dramatic progress of deep learning has sparked such a burst of activity that venture capitalists who did not even know what deep learning was all about some years back, today are suspicious of the startups that do not have it. In near future, deep learning may herald an age where it may assist humans in software development, science, and many more. Integrating deep learning with the whole toolbox of other artificial intelligence techniques may accomplish startling things that will have great impact in the field of technology.

1.5 How Deep Learning Works

Deep networks map input to target via a sequence of layered transformations, and that these layered transformations are learned by exposure to the training examples.

The transformations that a layer applies to its input are determined by the layer's weights, which are basically a bunch of numbers. In other words, transformations implemented by a layer are parameterized by its weights.

In this context, learning can be defined as the process of finding the values of the weights of all layers in the network in such a manner that input examples can be correctly mapped to their associated targets. A deep learning network contains thousands of parameters, and finding the right values of these parameters is not an easy task, particularly when the value of one parameter has an impact on the value of another parameter.

In order to train a deep network one needs to find out how far the calculated output of the network is from the desired value. This measure is obtained by using a loss function, also called as objective function. The objective function calculates the difference between the predicted output obtained from the network and the true target value for a specific example. This gives a measure of how well the network has learnt a specific example. The objective of the training is to find the values for the weights that minimize the chosen error function.

The difference obtained is then used as a feedback signal to adjust the weights of the network, in a way that loss score for the current example is lowered. This adjustment is done by the optimizer—backpropagation algorithm, the central algorithm in deep learning.

Backpropagation algorithm involves assigning random values to the weight vectors initially, so that the network just implements a series of random transformations. Initially, the output obtained from the network can be far from what it should be, and accordingly the loss score may be very high. With every example that is fed to the network, the weights are adjusted in such a direction that makes the loss score to decrease. This process is repeated a number of times, until the weight values that minimize the loss function are obtained. A network is said to have learned when the output values obtained from the network are as close as they can be to the target values.

1.6 Deep Learning Challenges

Deep learning networks have brought their own set of problems and challenges which outweighed the benefits of deep architectures for several decades. Training these architectures for general use was impractically slow. With limited computational power, deep learning networks were already overtaken by other approaches such as kernel methods. With the significant growth in computational power (particularly in GPUs and distributed computing) and access to large labeled datasets paved the way for its return.

However, despite the remarkable advances in this area, training deep models with a huge number of free parameters is an intricate and ill-posed optimization problem. Many research works have been dedicated to creating efficient training methods for deep architectures. The strategies reported in the literature that deal with the difficulties of training deep networks include developing better optimizers, using well-designed initialization strategies, using activation functions based on local competition and using skip connections between layers with the aim to improve the flow of information. However, deep network training still faces problems which are caused by the stacking of several nonlinear transformations and need to be addressed.

Moreover, deep learning involves using large amounts of data to learn progressively. While large amounts of data are available in many applications, however, in some areas copious amount of data are rarely available. More flexible models are required to achieve an enhanced learning ability when only a limited amount of data is available.

Deep learning networks are very good at solving one problem; however, using deep networks to solve a very similar problem requires retraining and reassessment. Although there are many advancements in this aspect, more work is required in developing deep learning models which can perform multitasks without the need of reworking on the whole architecture.

Bibliography

Bengio, Y.: Learning deep architectures for AI. Found. Trends Mach. Learn. **2**(1), 1–127 (2009)

Blaauw, M., Bonada, J.: A neural parametric singing synthesizer. arXiv preprint. arXiv:1704.03809

Cruz-Roa, A.A., Ovalle, J.E.A., Madabhushi, A., Osorio, F.A.G.: A deep learning architecture for image representation, visual interpretability and automated basal-cell carcinoma cancer detection. In: International Conference on Medical Image Computing and Computer-Assisted Intervention. Springer, Berlin, Heidelberg, pp. 403–410, Sept 2013

Deng, L., Yu, D.: Deep learning: methods and applications. Found. Trends Signal Process. **7**(34), 197–387 (2013)

Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep Learning. MIT Press, Cambridge (2016)

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.R., Jaitly, N., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97

Holden, D., Saito, J., Komura, T.: A deep learning framework for character motion synthesis and editing. ACM Trans. Graph. (TOG) **35**(4), 138:1–138:11

Kang, E., Min, J., Ye, J.C.: A deep convolutional neural network using directional wavelets for low-dose X-ray CT reconstruction. Med. Phys. **44**(10) (2017)

Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)

LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)

Shi, Y., Yao, K., Tian, L., Jiang, D.: Deep LSTM based feature mapping for query classification. In: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1501–1511 (2016)

Simonyan, K., Zisserman, A.: Two-stream convolutional networks for action recognition in videos. In: Advances in Neural Information Processing Systems, pp. 568–576 (2014)

Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint. arXiv:1409.1556

Taigman, Y., Yang, M., Ranzato, M.A., Wolf, L.: Deepface: closing the gap to human-level performance in face verification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1701–1708 (2014)

Vinyals, O., Toshev, A., Bengio, S., Erhan, D.: Show and tell: a neural image caption generator. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3156–3164 (2015)

Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., et al.: Google's neural machine translation system: bridging the gap between human and machine translation. arXiv preprint. arXiv:1609.08144

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., et al.: Show, attend and tell: neural image caption generation with visual attention. In: International Conference on Machine Learning, pp. 2048–2057, June 2015

Zhang, R., Isola, P., Efros, A.A.: Colorful image colorization. In: European Conference on Computer Vision. Springer, Cham, pp. 649–666, Oct 2016

Chapter 2
Basics of Supervised Deep Learning

2.1 Introduction

The use of supervised and unsupervised deep learning models has grown at a fast rate due to their success with learning of complex problems. High-performance computing resources, availability of huge amounts of data (labeled and unlabeled) and state-of-the-art open-source libraries are making deep learning more and more feasible for various applications. Since the main focus of this chapter is on supervised deep learning, Convolutional Neural Network (CNN or ConvNets) that is one of the most commonly used supervised deep learning models is discussed in this chapter.

2.2 Convolutional Neural Network (ConvNet/CNN)

Convolutional Neural Network also known as ConvNet or CNN is a deep learning technique that consists of multiple numbers of layers. ConvNets are inspired by the biological visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. Different neurons in the brain respond to different features. For example, certain neurons fire only in the presence of lines of a certain orientation, some neurons fire when exposed to vertical edges and some when shown horizontal or diagonal edges. This idea of certain neurons having a specific task is the basis behind ConvNets.

ConvNets have shown excellent performance on several applications such as image classification, object detection, speech recognition, natural language processing, and medical image analysis. Convolutional neural networks are powering core of computer vision that has many applications which include self-driving cars, robotics, and treatments for the visually impaired. The main concept of ConvNets is to obtain local features from input (usually an image) at higher layers and combine them into more complex features at the lower layers. However, due to its multilayered architecture, it is computationally exorbitant and training such networks on a large dataset

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_2

takes several days. Therefore, such deep networks are usually trained on GPUs. Convolutional neural networks are so powerful on visual tasks that they outperform almost all the conventional methods.

2.3 Evolution of Convolutional Neural Network Models

LeNet: The first practical convolution-based architecture was LeNet which used backpropagation for training the network. LeNet was designed to classify handwritten digits (MNIST), and it was adopted to read large numbers of handwritten checks in the United States. Unfortunately, the approach did not get much success as it did not scale well to larger problems. The main reasons for this limitation were as follows:

a. Small labeled datasets.
b. Slow computers.
c. Use of wrong nonlinearity (activation) function.

The use of appropriate activation function in a neural network has huge impact on the final performance. Any deep neural network that uses a nonlinear activation function like sigmoid or tanh and is trained using backpropagation suffers from *vanishing gradient*. Vanishing gradient is a problem found in training the neural networks with gradient-based training methods. Vanishing gradient makes it hard to train and tune the parameters of the top layers in a neural network. The problem worsens as the total number of layers in the network increases.

AlexNet: The first breakthrough came in 2012 when the convolutional model which was named AlexNet significantly outperformed all other conventional methods in ImageNet Large-Scale Visual Recognition Competition (ILSVRC) 2012 that featured the ImageNet dataset. The AlexNet brought down classification error rate from 26 to 15%, a significant improvement at that time. AlexNet was simple but much more efficient than LeNet. The improvements to overcome the above mentioned problems were due to the following reasons:

a. Large labeled image database (ImageNet), which contained around 15 million labeled images from a total of over 22,000 categories, was used.
b. The model was trained on high-speed GTX 580 GPUs for 5 to 6 days.
c. ReLU (Rectified Linear Unit) $f(x) = \max(x, 0)$ activation function was used. This activation function is several times faster than the conventional activation functions like sigmoid and tanh. The ReLU activation function does not experience the vanishing gradient problem.

AlexNet consists of five convolutional layers, three pooling layers, three fully connected layers, and a 1000-way softmax classifier.

ZFNet: In 2013, an improved version of CNN architecture called ZFNet was introduced. ZFNet reduced the filter size in the first layer from 11×11 to 7×7 and used a stride of 2 instead of 4 which resulted in more distinctive features and fewer

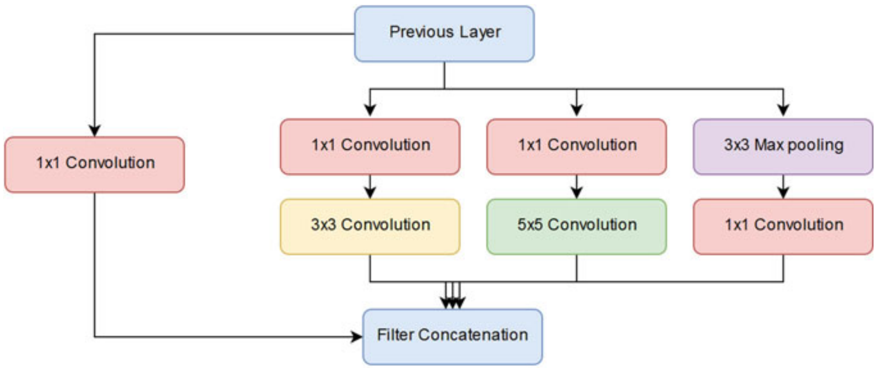


Fig. 2.1 Inception module in GoogLeNet

dead features. ZFNet turned out to be the winner of ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2013.

VGGNet: VGGNet, introduced in 2014, used increased depth of the network for improving the results. The depth of the network was made 19 layers by adding more convolutional layers with 3×3 filters, along with 2×2 max-pooling layers with stride and padding of 1 in all layers. Reducing filter size and increasing the depth of the network resulted in CNN architecture that produced more accurate results. VGGNet achieved an error rate of 7.32% in ILSVRC 2014 and was the runner-up model in ILSVRC 2014.

GoogLeNet: Google developed a ConvNet model called GoogLeNet in 2015. The model has 22 layers and was the winner of ILSVRC 2015 for having the error rate of 6.7%. The previous ConvNet models had convolution, and pooling layers stacked on top of each other but the GoogLeNet architecture is a little different. It uses an *inception* module which helps in reducing the number of parameters in the network. The *inception* module is actually a concatenated layer of convolutions (3×3 and 5×5 convolutions) and pooling sub-layers at different scales with their output filter banks concatenated into a single output vector making the input for the succeeding stage. These sub-layers are not stacked sequentially but the sub-layers are connected in parallel as shown in Fig. 2.1. In order to compensate for additional computational complexity due to extra convolutional operations, 1×1 convolution is used that results in reduced computations before expensive 3×3 and 5×5 convolutions are performed. GoogLeNet model has two convolutional layers, four max-pooling layers, nine inception layers, and a softmax layer. The use of this special inception architecture makes GoogLeNet to have 12 times lesser parameters than AlexNet.

Increasing the number of layers increases the number of features which enhances the accuracy of the network. However, there is a practical limitation to that. (1) Vanishing gradients: Some neurons in too deep networks may die during training which can cause loss of useful information and (2) Optimization difficulty: too many param-

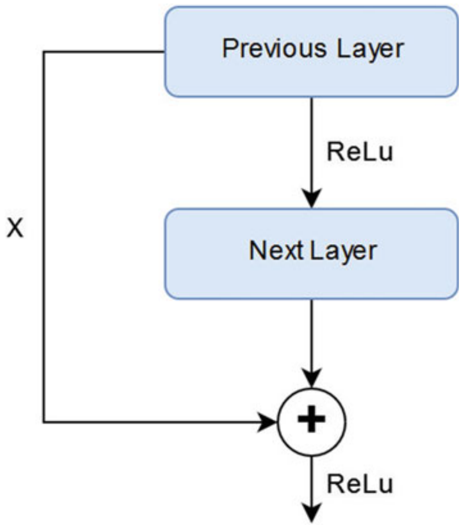


Fig. 2.2 Residual connection in ResNet

eters can make training the network a difficult task. The network depth should be increased without any negative effects. The inception model was refined as Inception V3 in 2016, and as Inception-ResNet in 2017.

ResNet: Microsoft Research Asia proposed a CNN architecture in 2015, which is, 152 layers deep and is called ResNet. ResNet introduced *residual connections* in which the output of a conv-relu-conv series is added to the original input and then passed through Rectified Linear Unit (ReLU) as shown in Fig. 2.2. In this way, the information is carried from the previous layer to the next layer and during backpropagation, the gradient flows easily because of the addition operations, which distributes the gradient. ResNet proved that a complex architecture like *Inception* is not required to achieve the best results but a simple and deep architecture can be tweaked to get better results. ResNet performed good in classification, detection, and localization and won ILSVRC 2015 with an incredible error rate of 3.6% which is better than the human error rate of 5–10%. ResNet is currently deepest network trained on ImageNet and has lesser parameters than VGGNet which is eight times lesser in depth.

Inception-ResNet: A hybrid inception model which uses residual connections, as in ResNet, was proposed in 2017. This hybrid model called Inception-ResNet dramatically improved the training speed of inception model and slightly outperformed the pure ResNet model by a thin margin.

Xception: A convolutional neural network architecture based on depthwise separable convolution layers is called Xception. The architecture is actually inspired by inception model and that is why it is called Xception (Extreme Inception). Xception architecture is a pile of depthwise separable convolution layers with residual connec-

Table 2.1 Classification accuracy of AlexNet, VGG-16, ResNet-152, Inception and Xception on ImageNet

Model	Top-1 accuracy	Top-5 accuracy
AlexNet	0.625	0.86
VGG-16	0.715	0.901
Inception	0.782	0.941
ResNet-152	0.870	0.963
Xception	0.790	0.945

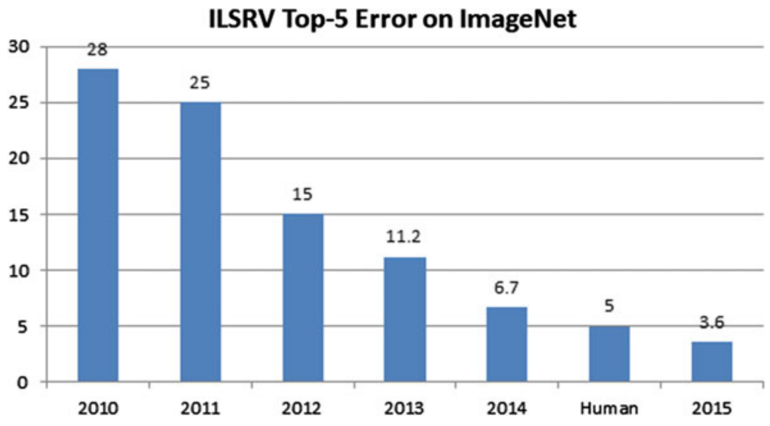


Fig. 2.3 ILSRV top-5 error on ImageNet since 2010

tions. Xception has 36 convolutional layers organized into 14 modules, all having linear residual connections around them, except for the first and last modules. The Xception has claimed to perform slightly better than Inception V3 on ImageNet. Table 2.1 and Fig. 2.3 show classification performance of VGG-16, ResNet-152, Inception V3 and Xception on ImageNet.

SqueezeNet: As the accuracy of new ConvNets models kept on improving, researchers started focusing on how to reduce the size and complexity of the existing ConvNet architectures without compromising on accuracy. The goal was to design a model that has very few parameters, while maintaining high accuracy. A pretrained model was used, and those of its parameters with values below a certain threshold were replaced with zeros to form a sparse matrix followed by few iterations of training on the sparse ConvNet.

Another version of SqueezeNet model used the following three main strategies to reduce the parameters and computational effort significantly while maintaining high accuracy. (a) Replace 3 × 3 filters with 1 × 1 filters. (b) Reduce the number of input channels to 3 × 3 filters. (c) Delay subsampling till late in the network so that convolution layers have large activation maps. SqueezeNet achieved AlexNet-level accuracy on ImageNet with 50 times fewer parameters.

ShuffleNet: Another ConvNet architecture called ShuffleNet was introduced in 2017 for devices with limited computational power, like mobile devices, without compromising on accuracy. ShuffleNet used two ideas, pointwise group convolution and channel shuffle, to considerably decrease the computational cost while maintaining the accuracy.

2.4 Convolution Operation

Convolution is a mathematical operation performed on two functions and is written as $(f * g)$, where f and g are two functions. The output of the convolution operation for domain n is defined as

$$(f * g)(n) = \sum_{m} f(m)g(n - m)$$

For time-domain functions, n is replaced by t. The convolution operation is commutative in nature, so it can also be written as

$$(f * g)(n) = \sum_{m} f(n - m)g(m)$$

Convolution operation is one of the important operations used in digital signal processing and is used in many areas which includes statistics, probability, natural language processing, computer vision, and image processing.

Convolution operation can be applied to higher dimensional functions as well. It can be applied to a two-dimensional function by sliding one function on top of another, multiplying and adding. Convolution operation can be applied to images to perform various transformations; here, images are treated as two-dimensional functions. An example of a two-dimensional filter, a two-dimensional input, and a two-dimensional feature map is shown in Fig. 2.4. Let the 2D input (i.e., 2D image) be denoted by A, the 2D filter of size $m \times n$ be denoted by K, and the 2D feature map be denoted by F. Here, the image A is convolved with the filter K and produces the feature map F. This convolution operation is denoted by $A*K$ and is mathematically given as

$$F(i, j) = (A * K)(i, j) = \sum_{m} \sum_{n} A(m, n)K(i - m, j - n) \tag{2.1}$$

The convolution operation is commutative in nature, so we can write Eq. 2.1 as

$$F(i, j) = (A * K)(i, j) = \sum_{m} \sum_{n} A(i - m, j - n)K(m, n) \tag{2.2}$$

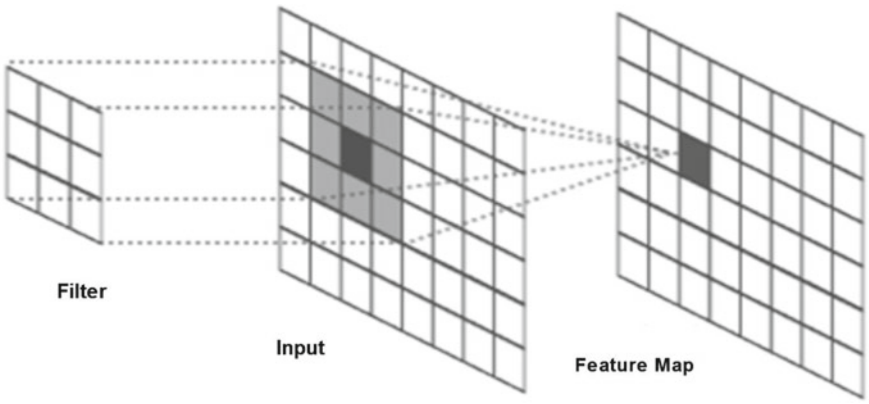


Fig. 2.4 Convolution operation

The kernel K is flipped relative to the input. If the kernel is not flipped, then convolution operation will be same as cross-correlation operation that is given below:

$$F(i, j) = (A * K)(i, j) = \sum_m \sum_n A(i + m, j + n)K(m, n) \tag{2.3}$$

Many CNN libraries use cross-correlation function as convolution function because cross-correlation is more convenient to implement than convolution operation itself. According to Eq. 2.3, the operation computes the inner product (element-wise multiplication) of the filter at every location in the image.

2.5 Architecture of CNN

In a traditional neural network, neurons are fully connected between different layers. Layers that sit between the input layer and output layer are called hidden layers. Each hidden layer is made up of a number of neurons, where each neuron is fully connected to all neurons in the preceding layer. The problem with the fully connected neural network is that its densely connected network architecture does not scale well to large images. For large images, the most preferred approach is to use convolutional neural network.

Convolutional neural network is a deep neural network architecture designed to process data that has a known, grid-like topology, for example, 1D time-series data, 2D or 3D data such as images and speech signal, and 4D data such as videos. ConvNets have three key features: local receptive field, weight sharing, and subsampling (pooling).

(i) **Local Receptive Field**

In a traditional neural network, each neuron or hidden unit is connected to every neuron in previous layer or every input unit. Convolutional neural networks, however, have local receptive field architecture, i.e., each hidden unit can only connect to a small region of the input called local receptive field. This is accomplished by making the filter/weight matrix smaller than the input. With local receptive field, neurons can extract elementary visual features like edges, corners, end points, etc.

(ii) **Weight Sharing**

Weight sharing refers to using the same filter/weights for all receptive fields in a layer. In ConvNet, since the filters are smaller than the input, each filter is applied at every position of the input, i.e., same filter is used for all local receptive fields.

ConvNet consists of a sequence of different types of layers to achieve different tasks. A typical convolutional neural network consists of the following layers:

- Convolutional layer,
- Activation function layer (ReLU),
- Pooling layer,
- Fully connected layer and
- Dropout layer.

These layers are stacked up to make a full ConvNet architecture. Convolutional and activation function layers are usually stacked together followed by an optional pooling layer. Fully connected layer makes up the last layer of the network, and the output of the last fully connected layer produces the class scores of the input image. In addition to these main layers mentioned above, ConvNet may include optional layers like batch normalization layer to improve the training time and dropout layer to address the overfitting issue.

(iii) **Subsampling (Pooling)**

Subsampling reduces the spatial size of the input, thus reducing the parameters in the network. There are few subsampling techniques available, and the most common subsampling technique is max-pooling.

2.5.1 Convolution Layer

Convolution layer is the core building block of a convolutional neural network which uses convolution operation (represented by *) in place of general matrix multiplication. Its parameters consist of a set of learnable filters also known as kernels. The main task of the convolutional layer is to detect features found within local regions of the input image that are common throughout the dataset and mapping their appearance to a feature map. A **feature map** is obtained for each filter in the layer by repeated application of the filter across subregions of the complete image, i.e., *convolving*

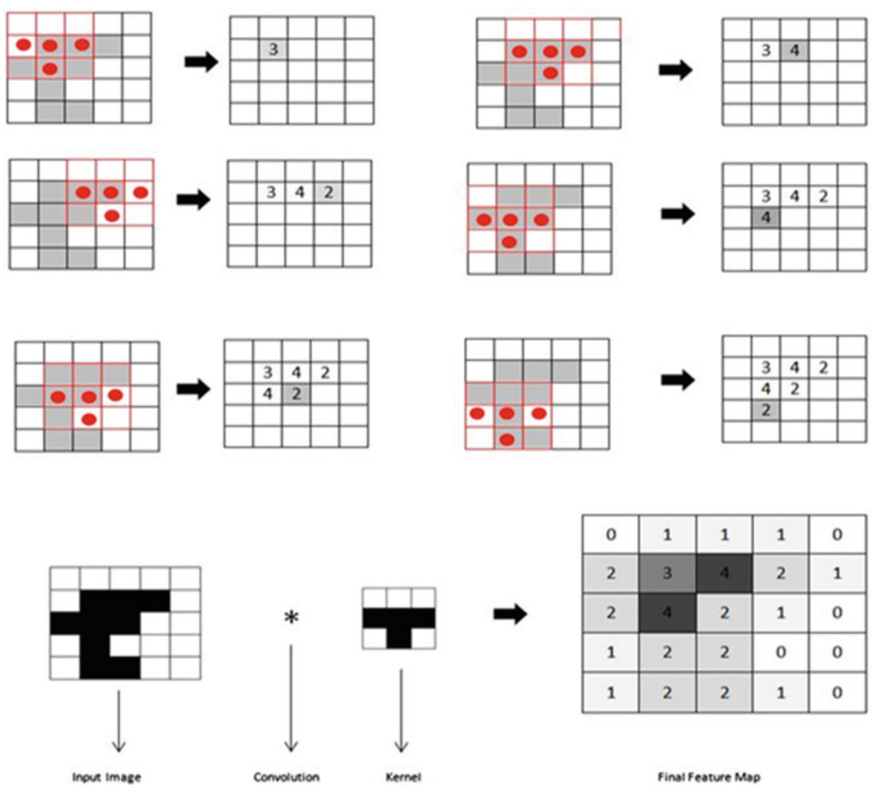


Fig. 2.5 Example of convolution operation

the filter with the input image, adding a bias term, and then applying an activation function. The input area on which a filter is applied is called **local receptive field**. The size of the receptive field is same as the size of the filter. Figure 2.5 shows how a filter (T-shaped) is convolved with the input to get the feature map.

Feature map is obtained after adding a bias term and then applying a nonlinear function to the output of the convolution operation. The purpose of nonlinearity function is to introduce nonlinearity in the ConvNet model, and there are a number of nonlinearity functions available which are briefly explained in the next section.

Filters/Kernels
The weights in each convolutional layer specify the convolution filters and there may be multiple filters in each convolutional layer. Every filter contains some feature like edge, corner, etc. and during forward pass, each filter is slid across the width and height of the input generating feature map of that filter.

Hyperparameters
Convolutional neural network architecture has many hyperparameters that are used to control the behavior of the model. Some of these hyperparameters control the size

of the output while some are used to tune the running time and memory cost of the model. The four important hyperparameters in the convolution layer of the ConvNet are given below:

a. *Filter Size*: Filters can be of any size greater than 2×2 and less than the size of the input but the conventional size varies from 11×11 to 3×3. The size of a filter is independent of the size of input.
b. *Number of Filters*: There can be any reasonable number of filters. AlexNet used 96 filters of size 11×11 in the first convolution layer. VGGNet used 96 filters of size 7×7, and another variant of VGGNet used 64 filters of size 11×11 in first convolution layer.
c. *Stride*: It is the number of pixels to move at a time to define the local receptive field for a filter. Stride of one means to move across and down a single pixel. The value of stride should not be too small or too large. Too small stride will lead to heavily overlapping receptive fields and too large value will overlap less and the resulting output volume will have smaller dimensions spatially.
d. *Zero Padding*: This hyperparameter describes the number of pixels to pad the input image with zeros. Zero padding is used to control the spatial size of the output volume.

Each filter in the convolution layer produces a feature map of size $([A - K + 2P]/S) + 1$ where A is the input volume size, K is the size of the filter, P is the number of padding applied and S is the stride. Suppose the input image has size 128×128, and 5 filters of size 5×5 are applied, with single stride and zero padding, i.e., $A = 128$, $F = 5$, $P = 0$ and $S = 1$. The number of feature maps produced will be equal to the number of filters applied, i.e., 5 and the size of each feature map will be $([128 - 5 + 0]/1) + 1 = 124$. Therefore, the output volume will be $124 \times 124 \times 5$.

2.5.2 Activation Function (ReLU)

The output of each convolutional layer is fed to an activation function layer. The activation function layer consists of an activation function that takes the feature map produced by the convolutional layer and generates the activation map as its output. The activation function is used to transform the activation level of a neuron into an output signal. It specifies the output of a neuron to a given input. An activation function usually has a squashing effect which takes an input (a number), performs some mathematical operation on it and outputs the activation level of a neuron between a given range, e.g., 0 to 1 or -1 to 1.

A typical activation function should be differentiable and continuous everywhere. Since ConvNets are trained using gradient-based methods, an activation function should be differential at any point. However, if a non-gradient-based method is used, then differentiability is not necessary.

There are many of activation functions in use with Artificial Neural Networks (ANNs) and some of the commonly used activation functions are as follows:

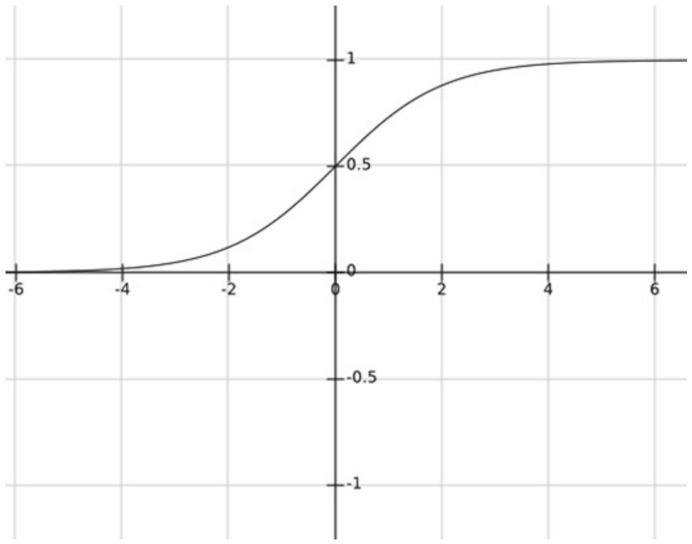


Fig. 2.6 Graph of sigmoid activation function

- **Logistic/Sigmoid Activation Function**: The sigmoid function is mathematically represented as

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

It is an S-shaped curve as shown in Fig. 2.6. Sigmoid function squashes the input into the range [0, 1].

- **Tanh Activation Function**: The hyperbolic tangent function is similar to sigmoid function but its output lies in the range $[-1, 1]$. The advantage of tanh over *sigmoid* is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph as shown in Fig. 2.7.
- **Softmax Function** (**Exponential Function**): It is often used in the output layer of a neural network for classification. It is mathematically represented as

$$\sigma\left(x_j\right) = \frac{\mathrm{e}^{x_j}}{\sum_{k=1}^{n} \mathrm{e}^{x_k}}$$

The softmax function is a more generalized logistic activation function which is used for multiclass classification.

- **ReLU Activation Function**: Rectified Linear Unit (ReLU) has gained some importance in recent years and currently is the most popular activation function for deep neural networks. Neural networks with ReLU train much faster than other

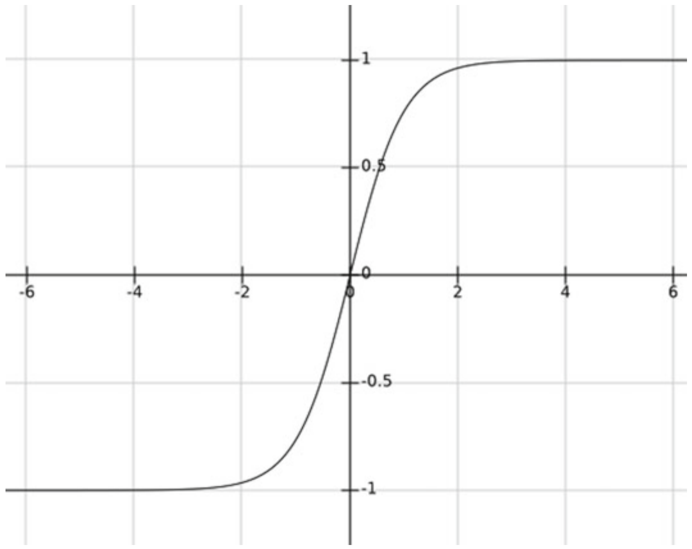


Fig. 2.7 Graph of tanh activation function

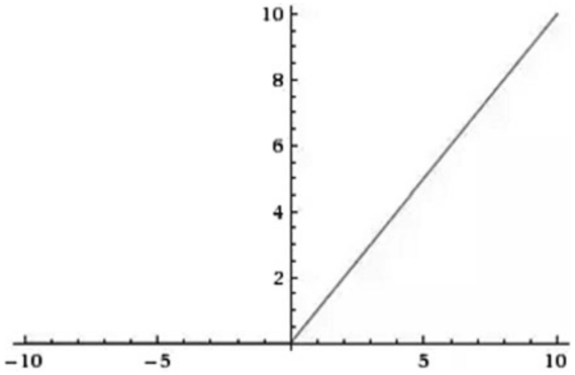


Fig. 2.8 Rectified Linear Unit (ReLU) activation function

activation functions like sigmoid and tanh. ReLU simply computes the activation by thresholding the input at zero. In other words, a rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. It is mathematically given as

$$\boldsymbol{f(x) = max(0, x)}$$

Rectified linear unit activation function produces a graph which is zero when $x < 0$ and linear with slope 1 when $x > 0$ as shown in Fig. 2.8.

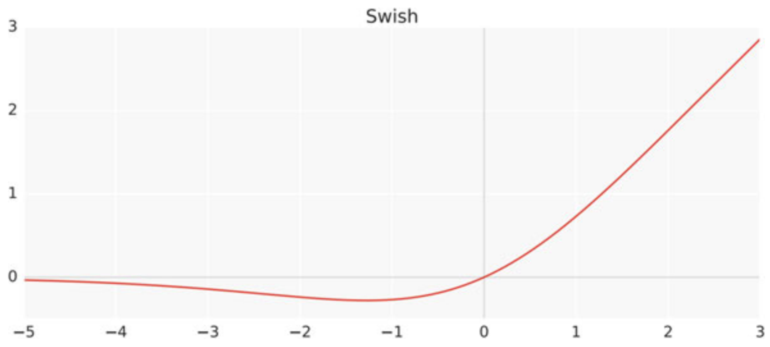


Fig. 2.9 The Swish activation function

- **SWISH Activation Function**:

Self-Gated Activation Function (SWISH) is actually a version of sigmoid function and is given as

$$f(x) = x * \sigma(x)$$

where $\sigma(x)$ is sigmoid of x given as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

SWISH activation function is a non-monotonic function and is shown in Fig. 2.9.

2.5.3 Pooling Layer

In ConvNets, the sequence of convolution layer and activation function layer is followed by an optional pooling or down-sampling layer to reduce the spatial size of the input and thus reducing the number of parameters in the network. A pooling layer takes each feature map output from the convolutional layer and down-samples it, i.e., pooling layer summarizes a region of neurons in the convolution layer. There are few pooling techniques available and the most common pooling technique is max-pooling. Max-pooling simply outputs the maximum value in the input region. The input region is a subset of input (usually 2×2). For example, if input region is of size 2×2, the max-pooling unit will output the maximum of the four values as shown in Fig. 2.10. Other options for pooling layers are average pooling and L2-norm pooling.

Pooling layer operation discards less significant data but preserves the detected features in a smaller representation. The intuitive reasoning behind pooling operation

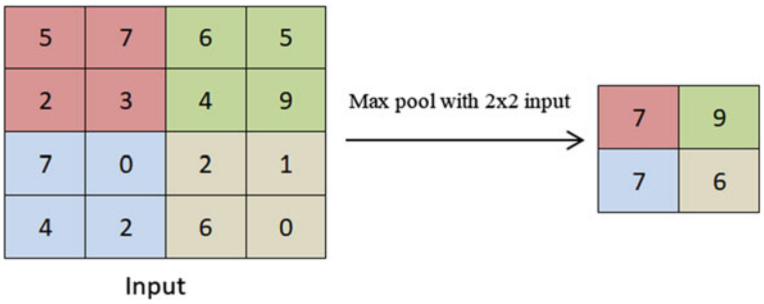


Fig. 2.10 Max-pooling

is that feature detection is more important than feature's exact location. This strategy works well for simple and basic problems but it has its own limitations and does not work well for some problems.

2.5.4 Fully Connected Layer

Convolutional neural networks are composed of two stages: Feature extraction stage and classification stage. In ConvNets, the stack of convolution and pooling layers act as feature extraction stage while as the classification stage is composed of one or more fully connected layers followed by a softmax function layer. The process of convolution and pooling continues until enough features are detected. Next step is to make a decision based on these detected features. In case of classification problem, the task uses the detected features in the spatial domain to obtain probabilities that these features represent each class, that is, obtain the class score. This is done by adding one or more fully connected layers at the end. In fully connected layer, each neuron from previous layer (convolution layer or pooling layer or fully connected layer) is connected to every neuron in the next layer and every value contributes in predicting how strongly a value matches a particular class. Figure 2.11 shows the connection between a convolution layer and a fully connected layer. Like convolutional layers, fully connected layers can be stacked to learn even more sophisticated combinations of features. The output of last fully connected layer is fed to a classifier which outputs the class scores. Softmax and Support Vector Machines (SVMs) are the two main classifiers used in ConvNets. Softmax classifier produces probabilities for each class with a total probability of 1, and SVM which produces class scores and the class having highest score is treated as the correct class.

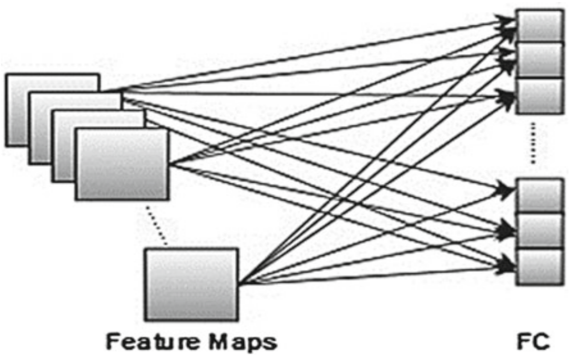


Fig. 2.11 Connection between convolution layer and fully connected layer

2.5.5 Dropout

Deep neural networks consist of multiple hidden layers enabling it to learn more complicated features. It is followed by fully connected layers for decision-making. A fully connected layer is connected to all features, and it is prone to overfitting. Overfitting refers to the problem when a model is trained and it works so well on training data that it negatively impacts the performance of the model on new data. In order to overcome the problem of overfitting, a dropout layer can be introduced in the model in which some neurons along with their connections are randomly dropped from the network during training (See Fig. 2.12). A reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights. Dropout notably reduces overfitting and improves the generalization of the model.

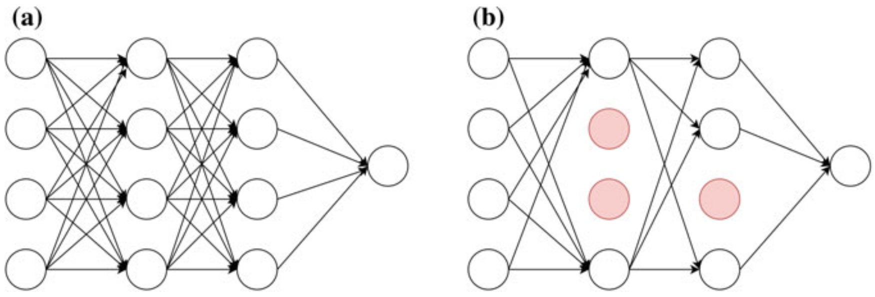


Fig. 2.12 **a** A simple neural network, **b** neural network after dropout

2.6 Challenges and Future Research Direction

ConvNets have evolved over the years and have achieved very good performance on various visual tasks like classification and object detection. In fact, deep networks have now achieved human-level performance in classifying different objects. However, deep networks like convolutional neural networks have their limitations. Altering an image in a way unnoticeable to humans can cause deep networks to miss-classify the image as something else. Modifying a few pixels selectively can make deep neural networks to produce incorrect results.

One of the reasons that make ConvNets vulnerable to these attacks is the way they accomplish pooling operation to achieve reduced feature space at the cost of losing important information about the precise location of the feature within the region. As a result, ConvNets can only identify if a certain feature exists in a certain region, irrespective of its position relative to another feature. The consequence is the difficulty in accurately recognizing objects that hold spatial relationships between features.

The vulnerabilities of deep networks put a big question mark on their reliability, raising questions about the true generalization capabilities of deep neural networks. A deep neural network architecture called ***Capsule Networks*** is used to address some of the inadequacies of ConvNets. A capsule network consists of capsules, which are a group of neurons representing the instantiation parameters of a specific object or part of it. Capsule networks use dynamic routing between capsules instead of max-pooling to forward information from layer to layer. The study on capsule networks is still in its early stages, and their performance on different visual tasks is not known yet.

Bibliography

1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
2. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
3. Nguyen, A., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: high confidence predictions for unrecognizable images. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2015, pp. 427–436
4. Ramachandran, P., Zoph, B., Le, Q.V.: Swish: a self-gated activation function. arXiv preprint arXiv: 1710.05941 (2017)
5. Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: Advances in Neural Information Processing Systems 2017, pp. 3859–3869
6. Salakhutdinov, R., Hinton, G.: Deep Boltzmann machines. In: Artificial Intelligence and Statistics, pp. 448–455 (2009)
7. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
8. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv: 1409.1556 (2014)

9. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
10. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: European Conference on Computer Vision, pp. 818–833. Springer, Cham (2014)

Chapter 3
Training Supervised Deep Learning Networks

3.1 Introduction

Training supervised deep learning networks involves obtaining model parameters using labeled dataset to allow the network to map an input data to a class label. The labeled dataset consists of training examples, where each example is a pair of an input data and a desired class label. The deep model parameters allow the network to correctly determine the class labels for unseen instances. This requires the model to generalize from the training dataset to unseen instances.

Many supervised deep learning models have been used by researchers. Convolutional neural network is one of the commonly used supervised deep architectures. This chapter will discuss the training of convolutional neural networks.

3.2 Training Convolution Neural Networks

Training supervised deep neural network is formulated in terms of minimizing a loss function. In this context, training a supervised deep neural network means searching a set of values of parameters (or weights) of the network at which the loss function has minimum value. Gradient descent is an optimization technique which is used to minimize the error by calculating gradients necessary to update the values of the parameters of the network.

The most common and successful learning algorithm for deep learning models is gradient descent-based backpropagation in which error is propagated backward from last layer to the first layer. In this learning technique, all the weights of a neural network are either initialized randomly or initialized by using probability distribution. An input is fed through the network to get the output. The obtained output and the desired output are then used to calculate the error using some cost function (error function). To understand the working of backpropagation, consider a small Convolution Neural Network (CNN) model shown in Fig. 3.1. The CNN model

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_3

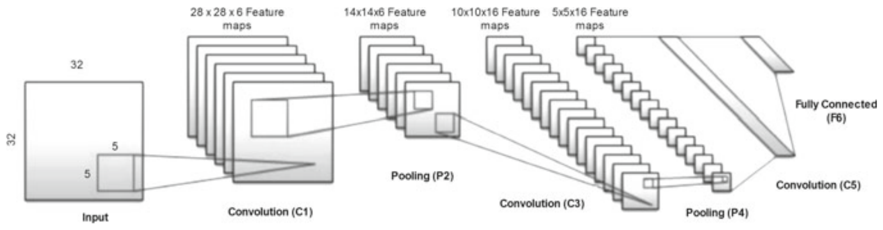


Fig. 3.1 CNN architecture

is similar to LeNet but uses ReLU (which has become a standard in deep networks) as activation function, max-pooling, instead of average pooling. The CNN model consists of three convolution layers, two subsampling layers, and one fully connected layer producing 10 outputs. The first convolution layer convolves input of size 32 × 32 with six 5 × 5 filters producing six 28 × 28 feature maps. Next, pooling layer down-samples these six 28 × 28 feature maps to size 14 × 14. Second convolution layer convolves the down-sampled feature maps with 5 × 5 filters producing 16 feature maps of size 10 × 10. Second pooling layer down-samples the input producing 16 feature maps of size 5 × 5. The third convolution layer has the input of size 5 × 5 and the filter of size 5 × 5, so no stride happens. We can say that this third layer has 120 filters of size 5 × 5 fully connected to each of the 16 feature maps of size 5 × 5. Last layer in the architecture is a fully connected layer containing 10 units for 10 classes. The output of the fully connected layer is fed to cost function (Softmax) to calculate the error. Let us describe the CNN architecture in detail including its mathematical operations.

Layer 1 (*C*1) is a convolution layer which takes input of size 32 × 32 and convolves it with six filters of size 5 × 5 producing six feature maps of size 28 × 28.

Mathematically, the above operation can be given as

$$C1^k_{i,j} = \left(\sum_{m=0}^{4} \sum_{n=0}^{4} w^k_{m,n} * I_{i+m,j+n} + b^k \right) \tag{3.1}$$

where $C1^k$ represent six output feature maps of convolution layer $C1$, k represents filter number, (m, n) are the indices of kth filter and (i, j) are the indices of output.

Equation 3.1 is the mathematical form of convolution operation. The output of convolution operation is fed to an activation function to make the output of linear operation nonlinear. In deep networks, the most successful activation function is ReLU given by

$$\sigma(x) = \max(0, x) \tag{3.2}$$

Using Eq. 3.1, we get

$$C1_{i,j}^k = \sigma\left(\sum_{m=0}^{4}\sum_{m=0}^{4} w_{m,n}^k * I_{i+m,j+n} + b^k\right) \tag{3.3}$$

Equation 3.3 is the output of layer $C1$.

Layer 2(*P*2) is a max-pooling layer. The output of the convolution layer $C1$ is fed to max-pooling layer. The max-pooling layer takes six feature maps from $C1$ and performs max-pooling operation on each of them.

The max-pooling operation on $C1^k$ is given as

$$P2^k = \text{MaxPool}\left(C1^k\right)$$

For each feature map in $C1^k$, max-pooling performs the following operation:

$$P2_{i,j}^k = \max\begin{pmatrix} C1_{(2i,2j)}^k, C1_{(2i+1,2j)}^k \\ C1_{(2i,2j+1)}^k, C1_{(2i+1,2j+1)}^k \end{pmatrix} \tag{3.4}$$

where (i, j) are the indices of kth feature map of output, and k is the feature map index.

Layer 3 (*C*3) is the second convolution layer which produces 16 feature maps of size 10 × 10 and is given by

$$C3_{i,j}^k = \sigma\left(\sum_{d=0}^{5}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P2_{i+m,j+n}^d + b^k\right)$$

where $C3^k$ represents the 16 output feature maps of convolution layer $C3$, k is the index of the output feature map, (m, n) are the indices of filter weights, (i, j) are the indices of output, and d is the index of the number of channels in the input.

Layer 4 (*P*4) is a max-pooling layer which produces 16 feature maps $P4^k$ of size 5 × 5.

The max-pooling operation is given as

$$P4^k = \text{MaxPool}\left(C3^k\right)$$

$$P4_{i,j}^k = \max\begin{pmatrix} C3_{(2i,2j)}^k, C3_{(2i+1,2j)}^k \\ C3_{(2i,2j+1)}^k, C3_{(2i+1,2j+1)}^k \end{pmatrix}$$

where (i, j) are the indices of kth feature map of output, and k is the feature map index.

Layer 5 (*C*5) is third convolution layer that produces 120 output feature maps and is given by

$$C5^k_{i,j} = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w^{k,d}_{m,n} * P4^d_{i+m,j+n} + b^k\right)$$

where $C5^k$ represents the 120 output feature maps of convolution layer $C5$ of size 1×1, k is the index of the output feature map, (m, n) are the indices of filter weights, d is the index of the number of channels in input and (i, j) are the indices of output, since output is only 1×1, the index (i, j) remains (0, 0) for each filter. This formula can be simplified as the filter size is equal to the size of input, so no convolution stride happens

$$C5^k = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w^{k,d}_{m,n} * P4^d_{m,n} + b^k\right)$$

Layer 6 (*F6*) is a fully connected layer. It consists of 10 neurons for 10 classes and is mathematically defined as

$$F6^k = \sum_{i=1}^{120} w^k_i * C5^i \tag{3.5}$$

In last layer, for each neuron, the activation function used is softmax function given as

$$Z^k = \mathrm{softmax}\left(F6^k\right)$$

The softmax function is defined as

$$Z^k = \mathrm{softmax}\left(F6^k\right) = \frac{\mathrm{e}^{F6^k}}{\sum_{i=1}^{10} \mathrm{e}^{F6^i}} \tag{3.6}$$

The softmax activation function produces the final output of the neurons in the range [0, 1] and all outputs add up to 1. Each of the output represents the probability of the input belonging to a particular class.

Here, Z^k is the vector of size 10 containing final output of the network.

Backward Pass

Loss Layer

During training, an input is fed to the network and output is obtained. The obtained output is compared against the actual output to calculate the error or loss. The calculated error is then used to update the weights of the network. The process is repeated until the error is minimized. The function which is used to calculate the error or loss is called cost/loss function, and there are quite a few loss functions available and for softmax layer the two commonly used loss functions are Mean Squared Error (MSE) and cross-entropy loss function. The Mean Squared Error (MSE) of the CNN model can be given as

$$\text{loss} = E(Z, \text{target})$$

The loss function E is defined as

$$E(Z, \text{target}) = \frac{1}{10}\sum_{k=1}^{10}\left(Z^k - \text{target}^k\right)^2 \tag{3.7}$$

Z^k is the kth output (in this case the network will produce 10 outputs representing class probabilities) generated by the CNN and target^k is the ground truth of the input.

$E(Z, \text{target})$ is the error/loss which represents how far the prediction of the network is from the actual target.

The purpose of training is to minimize the loss function and to do so the weights of the network are continuously updated until the minimum value is achieved. After calculating the error, the gradient of the loss function with respect to each weight of the neural network is calculated. The weights of the network are then updated with their respective gradients. This process is continued until the total error is minimized.

To minimize the loss function E, derivative of E is calculated w.r.t weight w^i. Since the loss function E is defined as composition of functions in series as

$$E = \text{loss}(\text{softmax}(F6(C5(P4(C3(P2(C1(input)))))))$$

That is, the loss function composes of the softmax activation function, which is a function of the fully connected layer $F6$ which in turn is a function of previous layer $C5$ and so on.

The loss function E is differentiable and can be differentiated w.r.t any weight at any layer using chain rule. In backpropagation, the weights of the last layer are updated first followed by second last layer and so on. To start with, let us find the derivative of the cost function *w.r.t* weight w_i^k at last layer $F6$ using chain rule.

$$\frac{\partial E}{\partial w_i^k} = \frac{\partial E}{\partial Z^k} * \frac{\partial Z^k}{\partial F6^k} * \frac{\partial F6^k}{\partial w_i^k} \tag{3.8}$$

- E is the cost function.
- Z^k represents the output of softmax function.
- $F6^k$ is the output of the last layer.
- k is the output layer neuron index.
- w_i^k is the ith weight of kth neuron of last layer.

After finding the derivative $\frac{\partial E}{\partial w_i^k}$, the weight w_i^k is then updated as

$$w_i^k = w_i^k - \mu\frac{\partial E}{\partial w_i^k} \tag{3.9}$$

where w_i^k is the updated weight and μ is the learning rate.

To solve Eq. 3.8, we first need to find the individual derivatives $\frac{\partial E}{\partial Z^k}$, $\frac{\partial Z^k}{\partial F6^k}$ and $\frac{\partial F6^k}{\partial w_i^k}$. which are given as

(i) $\frac{\partial E}{\partial Z^k} = \frac{\partial \frac{1}{10} \sum_{k=1}^{10} \left(Z^k - \text{target}^k\right)^2}{\partial Z^k} = \frac{1}{5}\left(Z^k - \text{target}^k\right)$

(ii) $\frac{\partial Z^k}{\partial F6^k} = Z^k * \left(1 - Z^k\right)$
Here, Z^k is the output of softmax function and the derivative of softmax function $f(x)$ is given as $f(x) * (1 - f(x))$ as shown in Sect. 3.3.3.

(iii) $\frac{\partial F6^k}{\partial w_i^k} = C5^i$

From Eq. 3.5

$$F6^k = \sum_{i=1}^{120} w_i^k * C5^i = \left(w_1^k * C5^1 + \cdots + w_i^k * C5^i + \cdots + w_{120}^k * C5^{120}\right)$$

The derivative of above function is given as

$$\frac{\partial F6^k}{\partial w_i^k} = \left(0 + \cdots + C5^i + \cdots + 0\right) = C5^i$$

Therefore,

$$\frac{\partial E}{\partial w_i^k} = \frac{1}{5}\left(Z^k - \text{target}^k\right) * Z^k * \left(1 - Z^k\right) * C5^i$$

Here, k is the neuron number in the last layer and i is the index of the weights to that neuron connecting input $C5^i$. For simplification in the backward pass and for propagation of error to previous layers, we calculate the deltas for this layer which is represented by

$$\delta F6^k = \frac{1}{5}\left(Z^k - \text{target}^k\right) * Z^k * \left(1 - Z^k\right)$$

These delta values enable the calculation of the gradient as well as the process of propagation of this error value to previous layers. The gradient then becomes as

$$\frac{\partial E}{\partial w_i^k} = \delta F6^k * C5^i$$

Here, i is the index of the weight to be updated and k is the neuron/filter number of layer $F6$.

The gradient $\frac{\partial E}{\partial w_i^k}$ calculated above can now be used to update the weight w_i^k using the formula

$$w_i^k = w_i^k - \mu \frac{\partial E}{\partial w_i^k}$$

Hidden Layer ($C5$)

The backward pass will continue to update the weights in hidden layer $C5$. The equations during forward pass at the layer are given as

$$C5^k = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P4_{m,n}^d + b^k\right)$$

where $C5^k$ represents the 120 feature maps of convolution layer $C5$, (m, n) are the indices of kth filter and d is the channel number.

In order to update the weight $w_{m,n}^{k,d}$, we first have to backpropagate the error from the layer $F6$ to layer $C5$ and then calculate the deltas of layer $C5$. The error backpropagated at this layer $C5$ from layer $F6$ is given by this

$$\mathrm{e}^k = \sum_{l=1}^{10} \delta F6^l * w_k^l$$

Here, w_k^l is the weight of the $F6$ layer connecting output of the layer $C5$ to the layer $F6$; k is the index of the 120 neurons in layer $C5$, and l is the index of delta vector at layer $F6$.

The deltas for this layer are given by the formula

$$\delta C5^k = \mathrm{e}^k * \mathrm{ReLU}'\left(x^k\right)$$

$$\mathrm{ReLU}'\left(x^k\right) = \left\{0 \text{ if } x^k < 0, 1 \text{ if } x^k \text{ otherwise}\right\}$$

Here, x^k is the summation of the inputs to the neuron multiplied with the weights of this layer; it is the input to the neuron before the activation function during the forward pass, and k is the index of the number of filter at this layer.

The gradient of any weight in this layer $C5$ is given by

$$\frac{\partial E}{\partial w_{m,n}^{k,d}} = \delta C5^k * P4_{m,n}^d$$

Here, d is the channel number and (m, n) are the indices of filter weights; k is the filter number in this layer $C5$.

Hidden Layer ($C3$)

To calculate the gradient of any of the weights in this layer, we have to backpropagate the error from layer $C5$ through pooling layer $P4$ to layer $C3$.

$$\mathrm{e}_{i,j}^{k} = \sum_{l=1}^{120} \delta C5^{l} * w_{i,j}^{l,k}$$

Here, $w_{i,j}^{l,k}$ is the weights of the next layer $C5$, l represents the filter number at that layer, k is the channel number/filter number connecting the output of layer $C3$ to $C5$, and (i, j) are the indices of the filter weights at $C5$, as well as the indices of the error matrix at $C3$ layer. Since we propagate error from $C5$ to $C3$ through $P4$ pooling layer, the error value is calculated for only those features which are selected during the max-pooling operation in forward pass.

This is used to calculate the delta values for this layer $C3$. These delta values are then used to calculate the gradient for any weight in this layer $C3$

$$\delta C3_{i,j}^{k} = \mathrm{e}_{i,j}^{k} * \mathrm{ReLU}'\left(x_{i,j}^{k}\right)$$

Here, $x_{i,j}^{k}$ is the summation of the summation of the inputs to the neuron multiplied with the weights of this layer; it is the result that is input to the activation function during the forward pass.

The gradient for the weights in this layer is calculated using the formula

$$\frac{\partial E}{\partial w_{m,n}^{k,d}} = \sum_{i=0}^{9} \sum_{j=0}^{9} \delta C3_{i,j}^{k} * P2_{m+i,n+j}^{d}$$

Here, $w_{m,n}^{k,d}$ is the weights of this layer, k is the filter number, d is the channel number, (m, n) are the indices of the weights and (i, j) are the indices of the delta matrix.

Hidden Layer ($C1$)

To calculate the gradient of any of the weights in this layer, we have to backpropagate the error from layer $C3$ through pooling layer $P2$ to layer $C1$. This operation is implemented by full convolution of the delta matrix with the 180° flipped weight matrix.

$$\mathrm{e}^{k} = \sum_{l=1}^{16} \delta C3^{l} * w^{l,k} (\text{flipped } 180^{\circ})$$

Here, e^{k} is the error matrix for each of the k filters; k goes from 1 to 6 as there are six filters in layer $C1$. The size of this error matrix is equal to the size of output of this layer, which is 28×28. The indices (i, j) for each error value of every matrix e^{k} go from 0 to 27. This is used to calculate the delta values for this layer $C1$. These delta values are then used to calculate the gradient for any weight in this layer $C1$

$$\delta C1_{i,j}^{k} = \mathrm{e}_{i,j}^{k} * \mathrm{ReLU}'\left(x_{i,j}^{k}\right)$$

Here, $x^k_{i,j}$ is the summation of the inputs to the neuron multiplied with the weights of this layer; it is the result that is input to the activation function during the forward pass, and (i, j) is the index of the delta matrix which goes from 0 to 27.

The gradient for the weights in this layer is calculated using the formula

$$\frac{\partial E}{\partial w^k_{m,n}} = \sum_{i=0}^{27}\sum_{j=0}^{27} \delta C1^k_{i,j} * \text{input image}_{m+i,n+j}$$

Here, $w^k_{m,n}$ is the weights of this layer, k is the filter number, (m, n) are the indices of the weights and (i, j) are the indices of the delta matrix.

3.3 Loss Functions and Softmax Classifier

A loss function is used to calculate the error (difference between prediction and ground truth label) during the training process of a deep network.

Depending upon the application, there are many loss functions that can be used in deep learning networks. For example, mean squared error ($L2$) loss, cross-entropy loss, and hinge loss are commonly used in classification problem. Absolute deviation error ($L1$) loss is suitable for regression problem. Some of the commonly used loss functions are discussed below.

3.3.1 Mean Squared Error (L2) Loss

The most commonly used loss function in machine learning is Mean Squared Error (MSE) loss function. The MSE function, also known as $L2$ loss function, calculates the squared average error E of all the individual errors, and is given by

$$E = \frac{1}{n}\sum_{i=1}^{n} \mathrm{e}_i^2$$

where e_i represents the individual error of ith output neuron which is given by

$$e_i = \text{target}(i) - \text{output}(i)$$

During training process, a loss function is used at the output layer to calculate the error and its derivative (gradient) is propagated in the backward direction of the network. The weights of the network are then updated with their respective gradients.

3.3.2 Cross-Entropy Loss

Cross-entropy loss is another loss function mostly used in regression and classification problems. Cross-entropy loss is given by

$$H(y) = -\sum_i y_i' \log(y_i)$$

where y_i' is the target label, and y_i is the output of the classifier. Cross-entropy loss function is used when the output is a probability distribution, and thus it is preferred loss function for softmax classifier.

3.3.3 Softmax Classifier

Softmax classifier is a mathematical function which takes an input vector and produces output vector in range (0–1), where the elements of the output vector add up to 1. That is, the sum of all the outputs of softmax function is 1. Softmax function is given by

$$S(y_i) = \frac{\mathrm{e}^{y_i}}{\sum_j \mathrm{e}^{y_j}}$$

Since softmax function outputs probability distribution, it is useful in the final layer of deep neural networks for multiclass classification.

During backpropagation, the derivative of this loss function is calculated using quotient rule

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = \frac{\left(\mathrm{e}^{y_i} \cdot \sum_{j=1}^{n} \mathrm{e}^{y_j}\right) - (\mathrm{e}^{y_i} \cdot \mathrm{e}^{y_i})}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2}$$

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = \frac{\left(\mathrm{e}^{y_i} \cdot \sum_{i}^{n} \mathrm{e}^{y_j}\right)}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2} - \frac{(\mathrm{e}^{y_i} \cdot \mathrm{e}^{y_i})}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2}$$

$$=> \frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = \frac{(\mathrm{e}^{y_i})}{\sum_{j=1}^{n} \mathrm{e}^{y_j}} - \frac{(e^{y_i} \cdot e^{y_i})}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2}$$

$$=> \frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = \frac{(\mathrm{e}^{y_i})}{\sum_{j=1}^{n} \mathrm{e}^{y_j}} - \left(\frac{\mathrm{e}^{y_i}}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)}\right)^2$$

$$=> \frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = S(y_i) - (S(y_i))^2$$

$$=> \frac{\mathrm{d}S(y_i)}{\mathrm{d}y_i} = S(y_i) \cdot (1 - (S(y_i))$$

$$\text{since } \frac{(\mathrm{e}^{y_i})}{\sum_{j=1}^{n} \mathrm{e}^{y_j}} = S(y_i)$$

Similarly, derivative of $S(y_i)$ w.r.t y_k is given as

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_k} = \frac{\left(0 \cdot \sum_{j=1}^{n} \mathrm{e}^{y_j}\right) - (\mathrm{e}^{y_i} \cdot \mathrm{e}^{y_k})}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2}$$

since $\frac{\mathrm{d}e^{y_i}}{\mathrm{d}y_k} = 0$ as e^{y_i} is constant here.

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_k} = -\frac{(e^{y_i} \cdot \mathrm{e}^{y_k})}{\left(\sum_{j=1}^{n} \mathrm{e}^{y_j}\right)^2}$$

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_k} = -\frac{\mathrm{e}^{y_i}}{\sum_{j=1}^{n} \mathrm{e}^{y_j}} \cdot \frac{\mathrm{e}^{y_k}}{\sum_{j=1}^{n} \mathrm{e}^{y_j}}$$

$$\frac{\mathrm{d}S(y_i)}{\mathrm{d}y_k} = -S(y_i) \cdot S(y_k)$$

The derivative of $S(y_i)$ is then used in Eq. 3.9 above to update the weights.

The gradient calculations are used to update the weights in such a way that the total error is minimized. The simplest way to minimize the error is to use various gradient-based optimization techniques which are briefly discussed in the next section.

3.4 Gradient Descent-Based Optimization Techniques

Gradient descent is an optimization technique used to minimize/maximize the cost function by calculating gradients necessary to update the values of the parameters of the network. Various variants of this optimizing technique define how to calculate the parameter updates using these gradients.

3.4.1 Gradient Descent Variants

There are three commonly used Gradient Descent (GD) variants. These variants differ in how many training examples are used to compute the gradient. The three variants are explained as follows.

3.4.1.1 Batch Gradient Descent (GD)

In traditional Gradient Descent (GD), also known as batch gradient descent, error gradient with respect to weight parameter w is computed for the entire training set followed by updating the weight parameter as shown below:

$$w = w - \mu \cdot \nabla \mathbf{E}(\mathbf{w})$$

where $\nabla \mathbf{E}(\mathbf{w})$ is the error gradient with respect to weight w and $\boldsymbol{\mu}$ is the learning rate that defines the step size to take along the gradient. The learning rate is a hyperparameter which cannot be too high or too low. Large value of learning rate can miss the optimum value, and too low learning rate will result in slow training time.

The training set often contains hundreds and thousands of examples that may demand huge memory which makes it difficult to fit in the memory. As a result, computing the error gradient can be very slow.

3.4.1.2 Stochastic Gradient Descent (SGD)

The above problem can be rectified by using Stochastic Gradient Descent (SGD), also known as incremental gradient descent, where gradient is computed for one training example at a time followed by updating of parameter values. It is usually much faster than standard gradient descent as it performs one update at a time.

$$w = w - \mu . \nabla \mathbf{E}(\mathbf{w}; \mathbf{x}(\mathbf{i}); \mathbf{y}(\mathbf{i}))$$

where $\nabla \mathbf{E}(\mathbf{w}; \mathbf{x}(\mathbf{i}); \mathbf{y}(\mathbf{i}))$ is the gradient of loss function—$E(w)$ w.r.t parameters w, for the training example $\{x(i), y(i)\}$.

In SGD, the one example based on updation of the parameter values causes the loss function to fluctuate frequently.

3.4.1.3 Mini-batch Gradient Descent

Mini-batch gradient descent also known as mini-batch SGD is a combination of both standard gradient descent and SGD techniques. Mini-batch SGD divides the entire training set into mini-batches of n training examples and performs the updating of parameter values for each mini-batch. This type of gradient descent technique takes advantage of both standard gradient descent and SGD techniques, and is commonly used optimization technique in deep learning.

$$w = w - \mu \cdot \nabla \mathbf{E}(\mathbf{w}; \mathbf{x}(\mathbf{i} : \mathbf{i} + \mathbf{n}); \mathbf{y}(\mathbf{i} : \mathbf{i} + \mathbf{n}))$$

Typical mini-batch size varies from 50 to 256 and should also be chosen sensibly according to the following factors:

- Large batch sizes provide more accurate gradients but have high memory requirements.
- Small batch sizes can offer a regularizing effect but require a small learning rate to maintain stability owing to the high variance in the estimate of the gradient. This in turn increases the training time because of the reduced learning rate.

3.4.2 *Improving Gradient Descent for Faster Convergence*

The main objective of optimization is to minimize the cost/loss or objective function. There are many methods available that help an optimization algorithm to converge faster. Some of the commonly used methods are discussed below.

3.4.2.1 AdaGrad

In SGD, the learning rate is set independently of gradients which may sometimes cause problems. For example, if the gradient is large, large learning rate would result in large step size, which means it may not achieve the optimum value as it may keep oscillating around the optimum value, and if the magnitude of gradient is small, a small learning rate may result in slow convergence. The problem can be resolved by using some adaptive approach for setting the learning rate. AdaGrad is one such adaptive model which uses adaptive learning rate by adding squared norms of previous gradients and dividing the learning rate by the square root of this sum.

$$w_{t+1,i} = w_{t,i} - \frac{\mu}{\sqrt{G_i}} \cdot \nabla_{t,i}$$

where $\nabla_{t,i}$ is the gradient of loss function with respect to parameter w_i and $G_i = \sum_{\tau=1}^{t} \nabla_{\tau}^2$ and ∇_τ is gradient at iteration τ.

In this way, parameters with high gradients will have small effective learning rate and parameters with small gradients receive increased effective learning rate.

The main advantage of AdaGrad is that learning rate is automatically adjusted, and there is no need to manually tune it. However, the sum in the denominator keeps on increasing which gradually causes the learning rate to decay. This decaying learning rate can slow down the learning or stop the learning completely.

3.4.2.2 AdaDelta

AdaDelta is a modified version of AdaGrad which overcomes the problem of decaying leaning rate. AdaDelta limits the number of previous gradients to some fixed size x and then the average of these past gradients is stored for efficiency. The average value $\mathrm{Avg}\big(\nabla_t^2\big)$ at time t only depends on previous average and current gradient. The parameter update is then made as

$$w_{t+1} = w_t - \frac{\mu}{\sqrt{\mathrm{Avg}\big(\nabla_t^2\big)}} \cdot \nabla_t$$

Since the denominator is just the Root Mean Square (RMS) of the parameters

$$w_{t+1} = w_t - \frac{\mu}{\mathrm{RMS}(\nabla_t)} \cdot \nabla_t$$

3.4.2.3 RMSProp

The vanishing learning rate in AdaGrad can be rectified by using RMSProp. It is a modified version of AdaGrad which discards history from the distant past by introducing exponentially weighted moving average. RMSProp uses sign of the gradient instead of the magnitude of the gradient to update the weights. The working of RMSProp optimizer is as follows:

(a) Set same magnitude of updates for all weights. Set maximum and minimum allowable weight updates to $\Delta_{\max}$ and $\Delta_{\min}$, respectively.

(b) At each iteration, if signs of current gradient and previous gradient are same, then increase learning rate by a factor of 1.2, i.e. $\eta = \eta + 1.2$.
Therefore, the update Δ_{ij}^{t+1} becomes

$$\Delta_{ij}^{t+1} = \min\big(\eta + \Delta_{ij}^{t}, \Delta_{\max}\big)$$

(c) If signs of current gradient and previous gradient are different, then decrease the learning rate by a factor of 0.5, i.e., $\eta = \eta - 0.5$

$$\Delta_{ij}^{t+1} = \max\big(\eta - \Delta_{ij}^{t}, \Delta_{\min}\big)$$

3.4.2.4 Adam

Adaptive Moment Estimation (Adam) is an adaptive optimization technique that takes advantages of AdaGrad and RMSProp. Like AdaDelta and RMSProp, Adam saves an exponentially decaying average of previous squared gradients v_t. In addition to that, Adam also computes the average of the second moments of the gradients m_t.

m_t and v_t which are values of the mean and uncentered variance, respectively, are given as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\boldsymbol{gt}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\boldsymbol{gt}^2$$

Adam updates exponential moving averages of the gradient and the squared gradient where the hyperparameters $\beta_1, \beta_2 \in [0, 1]$ control the decay rates of these moving averages.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The final formula for update is given as

$$w_{t+1} = w_t - \frac{\mu}{\sqrt{\hat{v}_t}} \cdot \hat{m}_t$$

Adam slightly performs better than other adaptive techniques and converges very fast. It also overcomes the problems faced by other optimization techniques such as decaying learning rate, high variance in updates, and slow convergence.

Note that a smoothing term 'e' is usually added in the denominator of the above fast converging methods to avoid divide by 0.

3.5 Challenges in Training Deep Networks

Training a deep neural network is a challenging task, and some of the prominent challenges in training deep models are discussed below.

3.5.1 Vanishing Gradient

Any deep neural network with activation function like sigmoid, tan*h,* etc. and training through backpropagation suffers from *vanishing gradient* problem. Vanishing gradient makes it very hard to train and update the parameters of the initial layers in the network. This problem worsens as the number of layers in the network increases. The aim of backpropagation in neural networks is to update the parameters such that the error of the network is minimized and actual output gets closer to the target output. During backpropagation, the weights are updated using gradient descent (rate

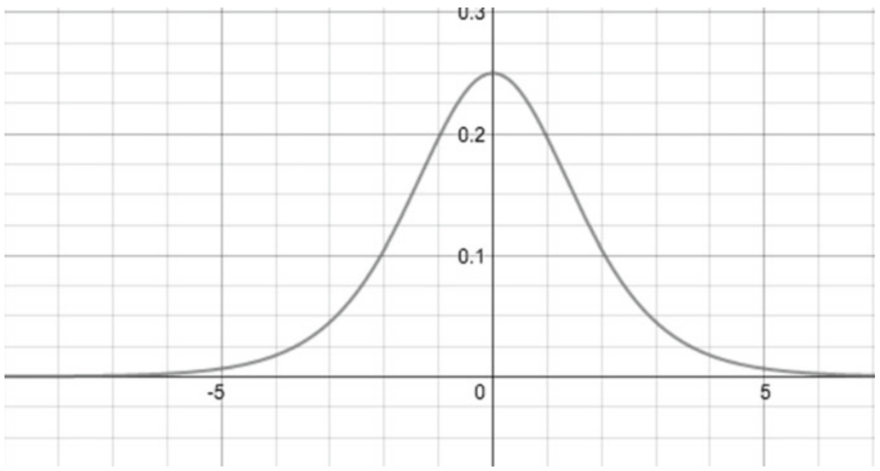


Fig. 3.2 Graph of derivative of sigmoid function

of change in total error E with respect to any weight w). In deep networks, these gradients determine how much each weight should change. The gradients become smaller as they propagate through many layers. The sigmoid function is given by

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

The derivative of this sigmoid function is given as

$$f'(x) = \frac{1}{1 + \mathrm{e}^{-x}}\left(1 - \frac{1}{1 + \mathrm{e}^{-x}}\right)$$

The graph of the above equation is given in Fig. 3.2. It is evident from the graph that the maximum point of the function is 0.25 implying that the output of the derivative of the cost function will always lie between 0 and 0.25. In other words, the errors will be squeezed to the range 0 and 0.25 at each layer. Therefore, the gradients become smaller and smaller after each layer and finally vanish leaving top layers untrained.

The vanishing gradient is the primary reason that makes sigmoid or tanh activations unsuitable for deep networks, and this is where Rectified Linear Units (ReLUs) come to the rescue. ReLU activation function does not suffer from vanishing gradient because there is no squeezing of inputs as the derivative is always 1 for positive inputs. A Rectified Linear Unit (ReLU) outputs 0 for input less than 0 and raw output otherwise. That is, if the input x is less than 0, then the output is 0 and if x is greater than 0, the output is equal to the input x and its derivative is 1.

That is $\boldsymbol{f(x) = x}$ and $\mathbf{f'(x) = 1}$ for $\mathbf{x > 0}$.

3.5.2 Training Data Size

Deep networks use training data for learning and are capable of learning complex nonlinear relationships between input data and output label. Deep networks require a large number of parameters to be learnt before it can be utilized to deliver the desired result. The number of parameters in deep models is large. More complex models mean more powerful abstraction, and more parameters which require more data. So, the size of training data is an important factor which can influence the success of deep models. In fact, all the successful deep models have been trained on some very large dataset. For example, AlexNet, GoogleNet, VGG, ResNet, etc. all have been trained on a vast dataset of images called ImageNet. ImageNet is an image dataset which contains around 1.2 million labeled images distributed over 1000 classes. However, one can argue that deep models for object recognition and detection require large number of parameters to deal with different variations, different poses, different variations in color, etc., and thus require vast size dataset for training. On the other hand, less complex problems (like classification of medical images) where the variations are very small as compared to variations mentioned above can be solved using less complex models which do not require huge training datasets. The claim is true to some extent, but model complexity alone cannot decide the size of the data required for training. Training data quality also plays an import role in it. Noisy data means low Signal-to-Noise Ratio (SNR) in the data and lower SNR means more data is required for convergence. Therefore, the size of dataset really depends on the complexity of the problem being studied, and the quality of data.

Irrespective of the complexity of the task, large training data size can significantly improve the performance of deep models. That is, the larger the training data size, the better the accuracy. But the question "How much data is enough?" still remains unanswered, and there is no rule of thumb that can define exact number of examples required to train a particular deep model.

3.5.3 Overfitting and Underfitting

Once a model is trained on a training dataset, it is expected to perform well on new, previously unseen data which was not present during learning. The ability of a machine learning model to perform well on new and unseen data is called **generalization**. Generalization is one of the objectives of a good deep learning model. To estimate the generalization ability of a deep learning model, it is tested on data collected separately from the training set.

Deep learning models can suffer from two problems, viz. overfitting and underfitting and both can lead to poor model performance.

Overfitting occurs when a model is trained, and it performs so well on training data that it is unable to generalize to new data. That is, the model has low training

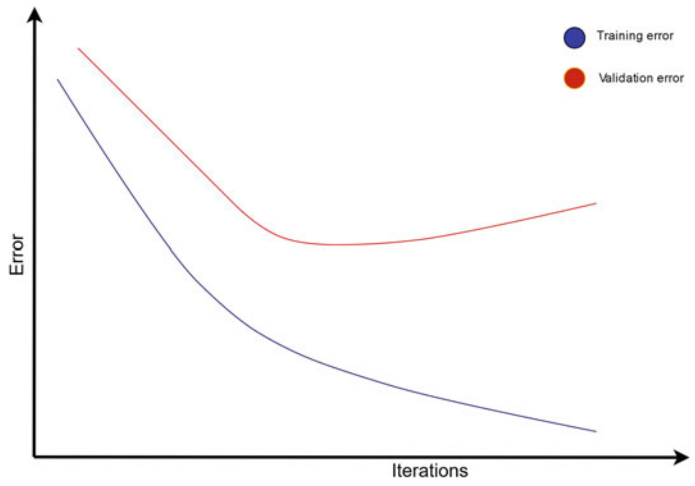


Fig. 3.3 Overfitting: training error (blue), validation error (red) as a function of the number of iterations

error but is unable to achieve low test error. In this case, the model is memorizing the data instead of learning. See Fig. 3.3.

Underfitting occurs when a model is not able to learn properly and its performance on the training set is poor.

The most common problem in deep learning is overfitting. Deep models like Convolutional Neural Network (ConvNet) have a large number of learnable parameters that must be learnt before they can be utilized to perform the task of interest. In order to train these models, extensively large training data with a specific end goal is required to accomplish the desired performance. If the training data is too small compared to the number of weights to be learned, then the network suffers from overfitting.

Overfitting is a common problem in deep networks, however, there are few techniques available that can be used in deep learning models to limit overfitting:

(a) Increase the training dataset.
(b) Reducing network size.
(c) Data augmentation: Modifying the current training data in a random way (Scaling, zooming, translation, etc.) to generate more training data.
(d) Interpolate weight penalties like $L1$ and $L2$ regularization and soft weight sharing.
(e) Dropout: The most popular technique to reduce overfitting is *Dropout*. Dropout refers to dropping out neurons/units in a neural network during training. Dropping a unit means temporarily detaching it from the network including all its

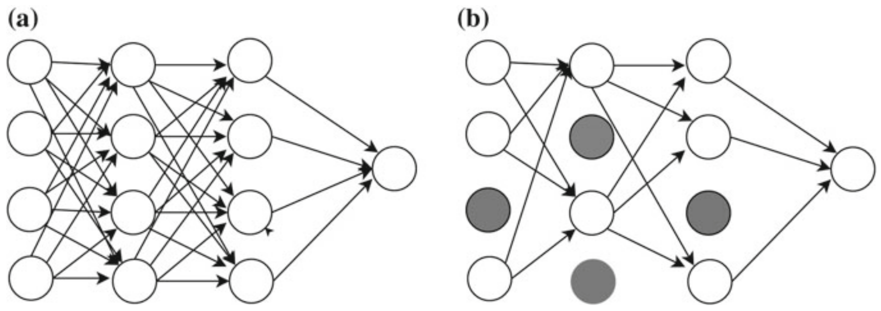


Fig. 3.4 **a** A simple neural network, **b** neural network after dropout

inward and outward connections (Fig. 3.4). The dropped-out neurons neither contribute to the forward pass nor do they contribute in backward pass. By using dropout, the network is forced to learn more robust features as network architecture changes with each input.

3.5.4 High-Performance Hardware

Training deep models on huge datasets require machines with sufficient processing power and memory. In order to get high efficiency and quick training time, it is highly recommended to use multi-core high-performance Graphics Processing Units (GPUs). These high-performance machines, GPUs, and memory are very costly and consume a lot of energy. Therefore, adopting deep learning solutions to real world becomes expensive and energy-consuming task.

3.6 Weight Initialization Techniques

One of the first tasks after designing a deep model is weight/parameter initialization. The weights which are learnt during training must have some initial values to start the training. Weight initialization is an important step which can have an intense effect on both the convergence rate and final accuracy of a network. Arbitrary initialization can lead to slow convergence or can completely freeze the learning process. A flawed or imperfect initialization of weights can impede the learning of a nonlinear system. Therefore, it is important to choose a robust technique to initialize the weights of a deep model. There are quite a few weight initialization techniques available each with its own weak features. Some initialization techniques are discussed below.

3.6.1 Initialize All Weights to 0

A simple way is to start from zero-valued weights and update these weights during training. This seems a sound idea but it has a problem associated with it. When all weights are initialized to 0, their derivative with respect to the loss function will be same for all weights. Thus, all the weights will have same value after successive iteration. This continues for all the training iterations making the model equivalent to a linear model.

3.6.2 Random Initialization

Another way is to initialize the weights with some random values (normal distribution) so that all weights are unique and compute distinct updates. This, however, suffers from two problems:

(a) Vanishing Gradients: If the weights are too small, their gradients will get smaller and smaller and finally vanish as they flow through different layers during back-propagation. This results in slow convergence and in worst case can freeze the learning process completely.
(b) Exploding Gradients: This is the opposite case of vanishing gradients. When the weights are too large, their gradients will be large as well causing large updates in the network weights. This results in unstable network as the weights keep oscillating around the minima. In worst case, the weights can become so large that overflow occurs and the output becomes NaN.

3.6.3 Random Weights from Probability Distribution

One good way to initialize weights is to assign the weights from a Gaussian distribution with zero mean and finite variance. With finite variance, the problem of vanishing and exploding gradients can be avoided. This type of technique in which weights are initialized such that variance remains same is called **Xavier initialization**.

For weight initialization, pick weights from a Gaussian distribution with zero mean and a variance of $1/N$, where N is the number of input neurons.

Xavier initialization is extensively used in neural networks with sigmoid activation function. Xavier initialization does not work well with ReLU activation function as it faces difficulties to converge. A modified version that uses

$$\text{var}(w_i) = \frac{2}{N}, \text{ instead of } \text{var}(w_i) = \frac{1}{N}$$

works well with ReLU activation function.

3.6.4 Transfer Learning

Another idea to initialize weights of a network is to transfer learnt weights from a trained model into the target model. In this way, no initialization actually takes place and the model gets previously learnt weights. This process of reusing or transferring weights learnt for one task into another similar task is called transfer learning. *Transfer learning thus refers to extracting the learnt weights from a trained base network (pretrained model) and transferring it to another untrained target network instead of training this target network from scratch.* In this way, features learned in one network are transferred and reused in another network designed to perform similar task. Transfer learning has gained popularity in deep learning especially on convolutional neural networks. It effectively reduces training time and also improves accuracy of models designed for tasks with minimum or inadequate training data. Transfer learning can be used in the following ways:

(a) *Pretrained model as fixed feature extractor*: In this scenario, the last fully connected layer (classifier layer) is replaced with a new linear classifier and this last layer is then trained on new dataset. In this way, the feature extraction layers remain fixed and only the classifier gets fine-tuned. This strategy is best suited when the new dataset is insufficient but similar to the original dataset.
(b) *Fine-tune whole Model*: Take a pretrained model, replace its last fully connected layer (classifier layer) with new fully connected layer and retrain the whole network on new dataset by continuing backpropagation up to the top layers. In this way, all the weights are fine-tuned for new task.

Transfer learning has many advantages and few drawbacks as well. Transfer learning is only possible when the two tasks have some amount of similarity. For totally different tasks, transfer learning may or may not work well. Furthermore, in order to use transfer learning, the two models should be compatible, i.e., base model and target model should have similar architecture.

3.7 Challenges and Future Research Direction

Many researchers have used deep networks and achieved good results in a wide range of applications. Despite this, application of deep networks in a given application poses many challenges and one of the challenges is training and optimization. Training deep models is not an easy task; just throwing raw training data and expecting that deep models will eventually learn by itself is not correct. Given right type of data and hyperparameters, a moderately simple model may perform well. But in general, learning an optimal deep model for a given application depends on various issues that need to be addressed carefully. Activation function, learning rates, weight initialization, data normalization, regularization, learning model structure, etc. all play an important role in the training process and to make appropriate selection or to choose an optimal value of a parameter is a challenging task. In addition, optimizing a cost function in general is still a difficult task. There are a number of challenges associated with neural network optimization problem and among these vanishing gradients, exploding gradients, local minima, flat regions, and local versus global structure are most common. In addition, optimization techniques are said to have many theoretical limitations as well which also needs to be explored.

Bibliography

Cho, J., Lee, K., Shin, E., Choy, G., Do, S.: How much data is needed to train a medical image deep learning system to achieve necessary high accuracy? arXiv preprint arXiv:1511.06348. 19 Nov 2015

Dauphin, Y.N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio, Y.: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in Neural Information Processing Systems, pp. 2933–2941 (2014)

Erickson, B.J., Korfiatis, P., Kline, T.L., Akkus, Z., Philbrick, K., Weston, A.D.: Deep learning in radiology: does one size fit all? J. Am. Coll. Radiol. **15**(3), 521–526 (2018)

Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. Book in preparation for MIT Press. URL http://www.deeplearningbook.org (2016)

He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1026–1034 (2015)

Kingma, D.P., Adam, J.B.: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 22 Dec 2014

Kumar, S.K.: On weight initialization in deep neural networks. arXiv preprint arXiv:1704.08863. 28 Apr 2017

Nowlan, S.J., Hinton, G.E.: Simplifying neural networks by soft weight-sharing. Neural Comput. **4**(4), 473–493 (1992)

Sussillo, D., Abbott, L.F.: Random walk initialization for training very deep feedforward networks. arXiv preprint arXiv:1412.6558. 19 Dec 2014

Wilson, D.R., Martinez, T.R.: The general inefficiency of batch training for gradient descent learning. Neural Netw. **16**(10), 1429–1451 (2003)

Chapter 4
Supervised Deep Learning Architectures

4.1 Introduction

Many supervised deep learning architectures have evolved over the last few years, achieving top scores on many tasks. Deep learning architectures can achieve high accuracy; sometimes, it can exceed human-level performance. Supervised training of convolutional neural networks, which contain many layers, is done by using a large set of labeled data. Some of the supervised CNN architectures proposed by researchers include LeNet-5, AlexNet, ZFNet, VGGNet, GoogleNet, ResNet, DenseNet, and CapsNet. These architectures are briefly discussed in this chapter.

4.2 LeNet-5

LeNet-5 is composed of seven layers which are fed by an input layer. The size of the input image used in LeNet-5 is 32 $\times$ 32 pixels. The values of the input pixels are normalized; as a result of this, the mean of the input tends to zero and variance roughly one, which accelerates the learning process. The architecture diagram of LeNet-5 is given in Fig. 4.1, and details of various layers are given in Table 4.1.

The first layer of LeNet-5 is the convolutional layer that produces six feature maps of size 28 $\times$ 28. There are 156 trainable parameters and 122,304 connections in the first convolutional layer.

The layer 2 performs down-sampling. There are 12 trainable parameters and 5880 connections in this layer. The third layer is a convolutional layer which produces 16 feature maps of size 10 $\times$ 10. The layer 4 is a subsampling layer that has 32 trainable parameters and 2000 connections.

The fifth layer is also a convolutional layer that produces 120 feature maps of size 1 $\times$ 1, and there are 48,120 connections in this layer. The layer 6 is a Fully Connected Layer (FCL) and has 84 outputs and 10,164 trainable parameters. The output layer

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_4

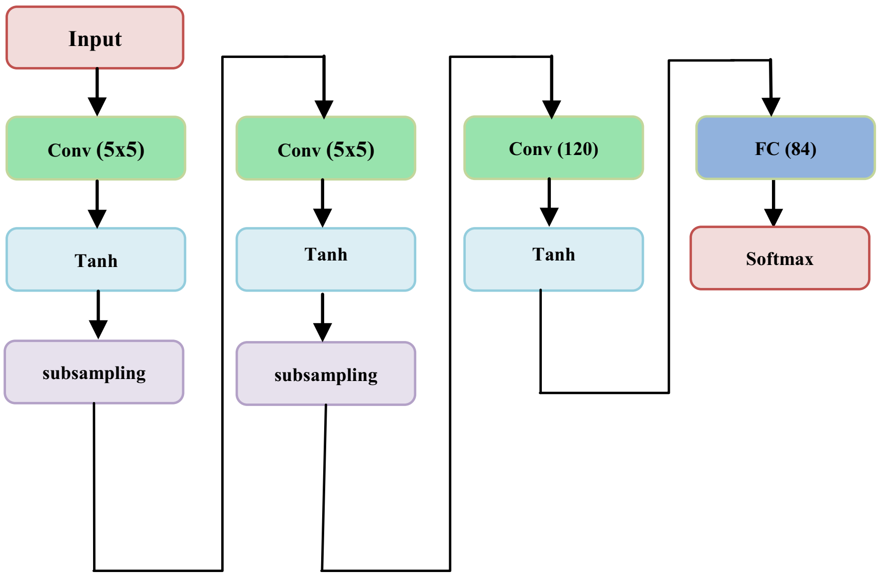


Fig. 4.1 Architecture diagram of LeNet-5

Table 4.1 Details of various layers of LeNet-5

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	32 × 32	5 × 5	–	6	1	0	28 × 28	6
Subsampling-1	28 × 28	–	2 × 2	–	2	0	14 × 14	6
Conv 2	14 × 14	5 × 5	–	16	1	0	10 × 10	16
Subsampling-2	10 × 10		2 × 2	16	2	0	5 × 5	16
Conv 3	5 × 5	5 × 5	–	120	1	0	1 × 1	120
Fully connected	120	–	–	–	–	–	1 × 1	84
Softmax	84	–	–	–	–	–	1 × 1	10

has radial basis function (Euclidean), one for each of the 10 output classes, and each output class is connected with 84 inputs.

The convolutional layers and the subsampling layers perform the task of feature extraction from the given image and the fully connected layers perform the task of classification. Figure 4.2 shows the handwritten digit "7" as input image. The target probability is 1 for the digit "7" and 0 for all the remaining nine digits. Therefore, the value of the target vector for digit "7" is given as [0, 0, 0, 0, 0, 0, 1, 0, 0]. Backpropagation is used to calculate the gradient of the loss function relating to all the weights in all layers of the CNN network.

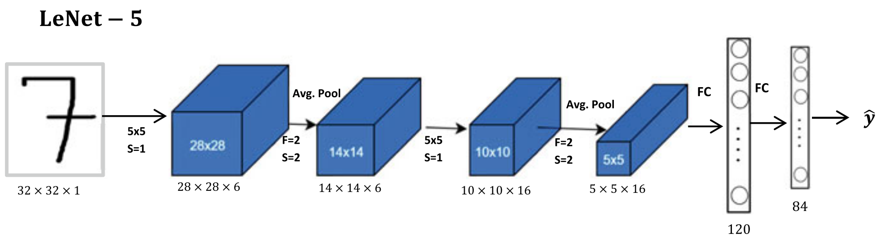


Fig. 4.2 Feature maps at various layers of LeNet-5

Training of LeNet-5:

Step 1: Random values are used for initialization of all filter parameters and weights.
Step 2: During this step, the input image goes through various layers, i.e., convolutional layers, subsampling layers, and fully connected layers. This step performs forward propagation which finds the output probabilities for all the classes in the network.
Step 3: The total error between output probabilities and target probabilities is calculated at the output layer of the network during this step.
Step 4: During this step, the error gradients with respect to weights in the network are calculated and gradient descent algorithm is used to update all weights and filter parameters to minimize the output error. The weights are adjusted proportionately depending on their contribution to the total error. Only the values of the connection weights and filter matrix are updated. The hyper-parameters like filter sizes and number of filters of the network are fixed and do not change during the training process.
Step 5: Steps 2–4 are repeated with all images present in the training set.

As shown in Table 4.1, the input image of size 32 × 32 is convolved with a filter of size 5 × 5 to generate six feature maps of size 28 × 28. Similarly, pooling is applied at layer 2 which down-samples the feature map of the first layer to 14 × 14. These feature maps of size 14 × 14 are again convolved with 16 filters of size 5 × 5 resulting in 16 feature maps of size 10 × 10. The result is again down-sampled to feature maps of size 5 × 5, and these feature maps are passed to a convolutional layer having a filter of size 5 × 5, which generates 120 feature maps of size 1 × 1. The output is then passed to a fully connected layer having 84 output neurons. The final classification layer in the network has ten neurons for ten classes.

4.3 AlexNet

One of the problems associated with training a deep neural network is vanishing gradient problem. This problem can be addressed if an activation function like Rectified Linear Unit (ReLU) is used. AlexNet is the first network which has used Rectified Linear Unit (ReLU) activation function.

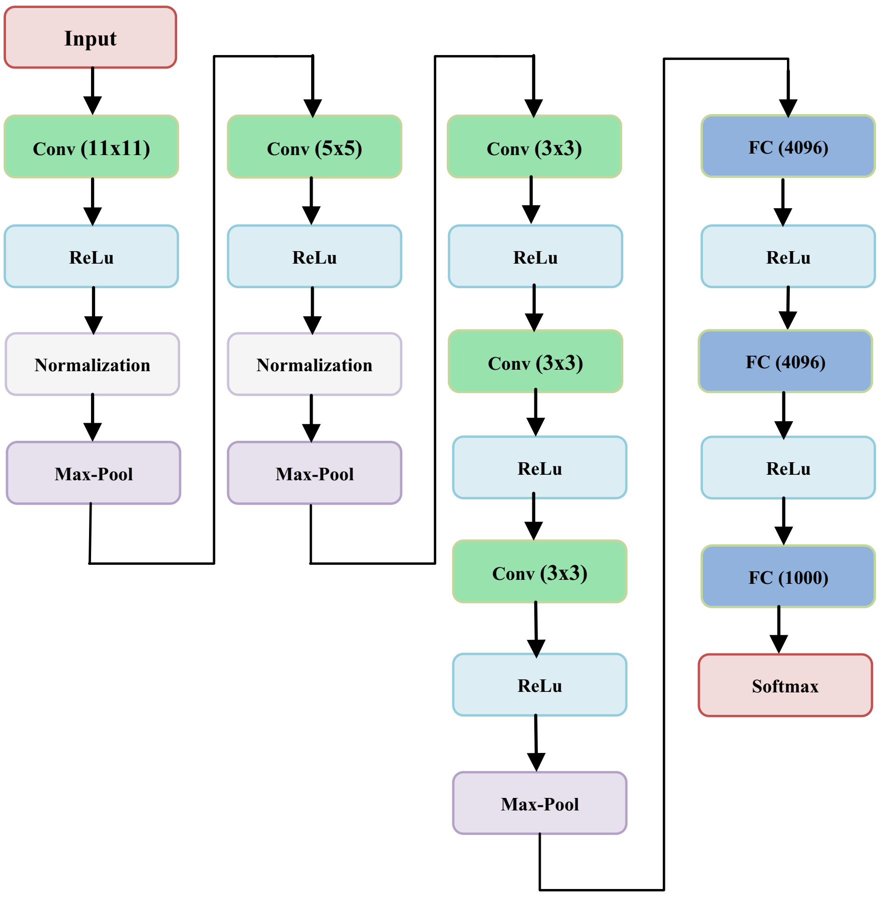


Fig. 4.3 Architecture diagram of AlexNet

The performance of AlexNet has been evaluated on ImageNet database. This database contains around 15 million high-resolution-labeled images of 22,000 subjects. AlexNet was trained on a subset of ImageNet database of around 1.2 million images of 1000 classes and tested on 150,000 images belonging to 1000 classes. The network was trained on GTX 580 GPU with 3 GB memory.

AlexNet is composed of many layers of convolutional, max-pooling, and fully connected layers. The first layer is convolutional layer, and the output of this layer is 96 images of size 55×55. The second layer is a max-pooling layer with output of size $27 \times 27 \times 96$ produced by using a window of size 3×3 with a stride of 2. The third layer is a convolutional layer with 256 filters of size 5×5 used with a stride of 1 and padding of 2. Layer 4 is a down-sampling layer also called max-pooling which produces an output of size $13 \times 13 \times 256$ by using a window of size 3×3 with a stride of 2. The layer 5 of this network is a convolutional layer that outputs

Table 4.2 Details of various layers of AlexNet

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	224 × 224	11 × 11	–	96	4	1	55 × 55	96
Max-pooling 1	55 × 55	–	3 × 3	–	2	0	27 × 27	96
Conv 2	27 × 27	5 × 5	–	256	1	2	27 × 27	256
Max-pooling 2	27 × 27	–	3 × 3	–	2	0	13 × 13	256
Conv 3	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 4	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 5	13 × 13	3 × 3	–	256	1	1	13 × 13	256
Max-pooling 3	13 × 13	–	3 × 3	–	2	0	6 × 6	256
Fully connected 1	4096 neurons							
Fully connected 2	4096 neurons							
Fully connected 3	1000 neurons							
Softmax	1000 Classes							

the feature maps of size 13 × 13 × 384 by using 384 filters of size 3 × 3 with a stride of 1 and a padding of 1. The sixth layer of this network is a convolutional layer that uses 384 filters of size 3 × 3 with a stride of 1 and padding of 1. Layer 7 is a convolutional layer that uses 256 filters of size 3 × 3, with a stride of 1, and padding of 1. The eighth layer is a down-sampling/max-pooling layer that uses a window of size 3 × 3 with a stride of 2. Layer 9 is the first fully connected layer with 4096 neurons. Layer 10 is a second fully connected layer with 4096 neurons. The eleventh layer of this network is a third fully connected layer with 1000 neurons. Final layer is a softmax layer. Figure 4.3 shows the architecture diagram of AlexNet, and the details of various layers of the AlexNet are given in Table 4.2.

AlexNet takes the input image of size 224 × 224, and the same is passed through five convolving layers, three fully connected layers, and then finally through softmax layer for classification. Using purely supervised learning, AlexNet showed that a large deep convolutional neural network can achieve good results on highly challenging datasets. No unsupervised pretraining was used in AlexNet.

4.4 ZFNet

ZFNet has the same architecture as that of AlexNet, but with smaller convolutional kernels of size 7×7 instead of size 11×11 used in AlexNet. The smaller convolutional kernels helped in obtaining better hyperparameters than that obtained by AlexNet and that too with less computational efforts.

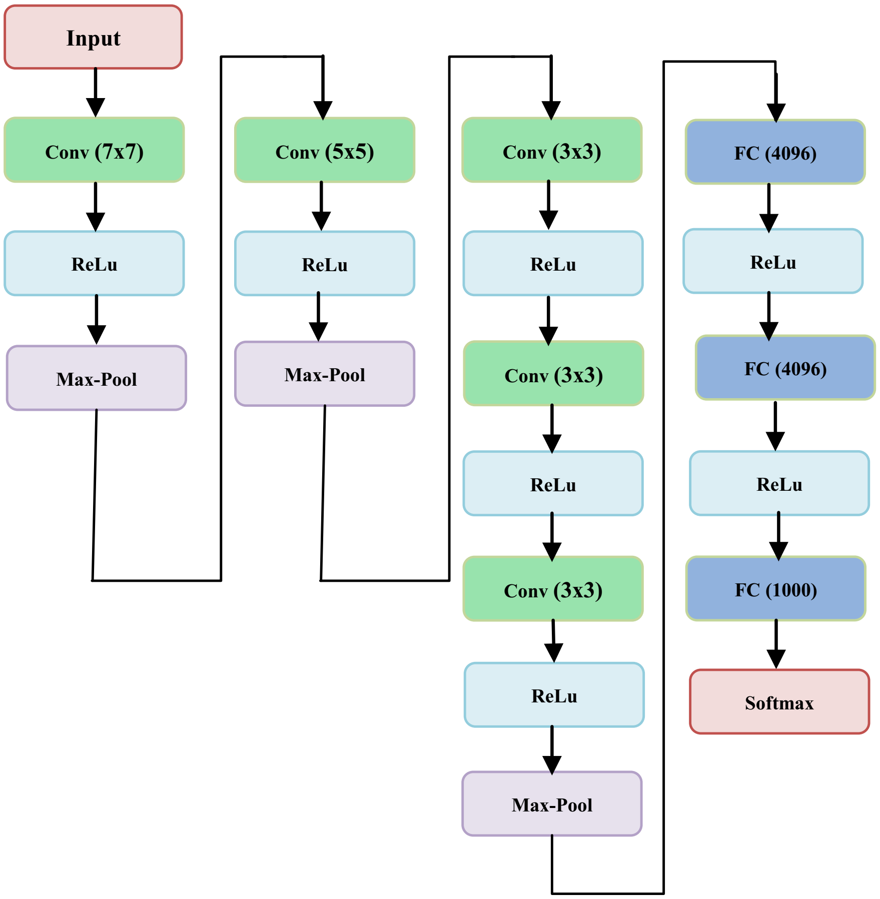


Fig. 4.4 Architecture diagram of ZFNet

Table 4.3 Details of various layers of ZFNet

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	224 × 224	7 × 7	–	96	2	0	110 × 110	96
Max-pooling 1	110 × 110	–	3 × 3	–	2	0	55 × 55	96
Conv 2	55 × 55	5 × 5	–	256	2	0	26 × 26	256
Max-pooling 2	26 × 26	–	3 × 3	–	2	0	13 × 13	256
Conv 3	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 4	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 5	13 × 13	3 × 3	–	256	1	1	13 × 13	256
Max-pooling 3	13 × 13	–	3 × 3	–	2	0	6 × 6	256
Fully connected 1	4096 neurons							
Fully connected 2	4096 neurons							
Fully connected 3	1000 neurons							
Softmax	1000 Classes							

ZFNet introduced an innovative visualization method called deconvolution. Deconvolution is a process that uses filtering and pooling in the reverse order of the forward pass so that input pixel space can be reconstructed from the intermediate feature maps.

The visualization technique was used to analyze the feature maps of the first and the second layers of the AlexNet architecture; the issues identified there were improved in ZFNet. The first layer filters produced appreciable low- and high-frequency information but negligible middle frequencies information. The visualization of the second layer feature maps indicated aliasing artifacts, which was caused by the large stride of 4 used in the first convolutional layer. To address these issues, the filter size in the first layer of ZFNet was reduced from 11 × 11 to 7 × 7 and the stride of the convolution was decreased from 4 to 2. This helped the architecture to retain more information in the feature maps of the first layer and the second layer, which improved the classification performance.

ZFNet used ReLU activation function, cross-entropy loss error function, and batch stochastic gradient descent for training. It was trained on GTX 580 graphical processing unit (GPU). Figure 4.4 shows architecture diagram used in ZFNet and Table 4.3 shows details of various layers of ZFNet.

The ZFNet has a total of five convolutional layers, three down-sampling layers, three fully connected layers, and a softmax layer as shown in Fig. 4.4.

Table 4.3 shows an input image of size 224 × 224 is passed through various layers indicating filter size, stride, and padding values for the classification task.

4.5 VGGNet

VGGNet has more layers than ZFNet and it was made feasible by using very small convolutional filters of size 3 × 3 in all layers. This improved the accuracy of VGGNet significantly as compared to ZFNet architecture.

AlexNet used a large receptive field of 11 × 11 with a stride of 4 in the first convolutional layer. This receptive field was reduced in ZFNet to 7 × 7 with a stride of 2 in the first convolutional layer. VGGNet proposed the use of a very small receptive field of 3 × 3 with stride of 1 throughout the whole network. Two convolution layers with receptive field of 3 × 3 without a spatial pooling in between have an effective receptive field of 5 × 5, and three such layers have an effective receptive field of 7 × 7. The use of three 3 × 3 convolutional layers is better than one 7 × 7 convolutional layer because of the two reasons: (i) The three 3 × 3 convolutional layers make use of three ReLU functions instead of one ReLU function used in one 7 × 7 convolutional layer. This makes decision function to be more discriminative. (ii) The number of parameters is decreased assuming that the number of channels used in the three convolutional layers is same.

The number of filters in case of VGGNet doubles after each max-pooling layer; it strengthens the objective of reduction in spatial dimensions but increases the depth. No Local Response Normalization (LRN) is used in this network. Other important characteristic of VGGNet is that it uses scale jittering during training which is one of the data augmentation techniques. The size of the input image to the VGG network is 224 × 224 RGB image and this size is fixed. The preprocessing in this network involves subtracting the mean RGB value, calculated using the training set, from each pixel. The value of padding is set to 1 for all 3 × 3 convolutional layers. Five max-pooling layers are used for spatial down-sampling. Max-pooling is performed using a 2 × 2 window with a stride of 2. The number of convolutional layers used in various VGG variants is different. The VGG-16 variant has 13 convolutional layers, 5 subsampling layers, and 3 fully connected layers. The first two fully connected layers have 4096 neurons each, and the last fully connected layer has 1000 neurons for 1000 classes. All the convolutional layers of VGGNet use the ReLU activation function. Figure 4.5 shows the architecture diagram of VGGNet-16, and Table 4.4 shows details of various layers of VGGNet-16.

Table 4.4 Details of various layers of VGGNet-16

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	224 × 224	3 × 3	–	64	1	1	224 × 224	64
Conv 2	224 × 224	3 × 3	–	64	1	1	224 × 224	64
Max-pooling 1	224 × 224	–	2 × 2	–	2	0	112 × 112	64
Conv 3	112 × 112	3 × 3	–	128	1	1	112 × 112	128
Conv 4	112 × 112	3 × 3	–	128	1	1	112 × 112	128
Max-pooling 2	112 × 112	–	2 × 2	–	2	0	56 × 56	128
Conv 5	56 × 56	3 × 3	–	256	1	1	56 × 56	256
Conv 6	56 × 56	3 × 3	–	256	1	1	56 × 56	256
Conv 7	56 × 56	3 × 3	–	256	1	1	56 × 56	256
Max-pooling 3	56 × 56	–	2 × 2	–	2	0	28 × 28	256
Conv 8	28 × 28	3 × 3	–	512	1	1	28 × 28	512
Conv 9	28 × 28	3 × 3	–	512	1	1	28 × 28	512
Conv 10	28 × 28	3 × 3	–	512	1	1	28 × 28	512
Max-pooling 4	28 × 28	–	2 × 2	–	2	0	14 × 14	512
Conv 11	14 × 14	3 × 3	–	512	1	1	14 × 14	512
Conv 12	14 × 14	3 × 3	–	512	1	1	14 × 14	512
Conv 13	14 × 14	3 × 3	–	512	1	1	14 × 14	512
Max-pooling 5	14 × 14	–	2 × 2	–	2	0	7 × 7	512
Fully connected 1	4096 neurons							

(continued)

Table 4.4 (continued)

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Fully connected 2	4096 neurons							
Fully connected 3	1000 neurons							
Softmax	1000 Classes							

This network performed well on localization as well as image classification tasks. The architecture was trained on four NVIDIA Titan Black Graphical Processing Units (GPUs).

Table 4.4 shows that an input image of size 224 $\times$ 224 is passed through various layers indicating filter size, stride, and padding values for the classification task.

The VGGNet has many variants like VGGNet-16, VGGNet-19, etc.

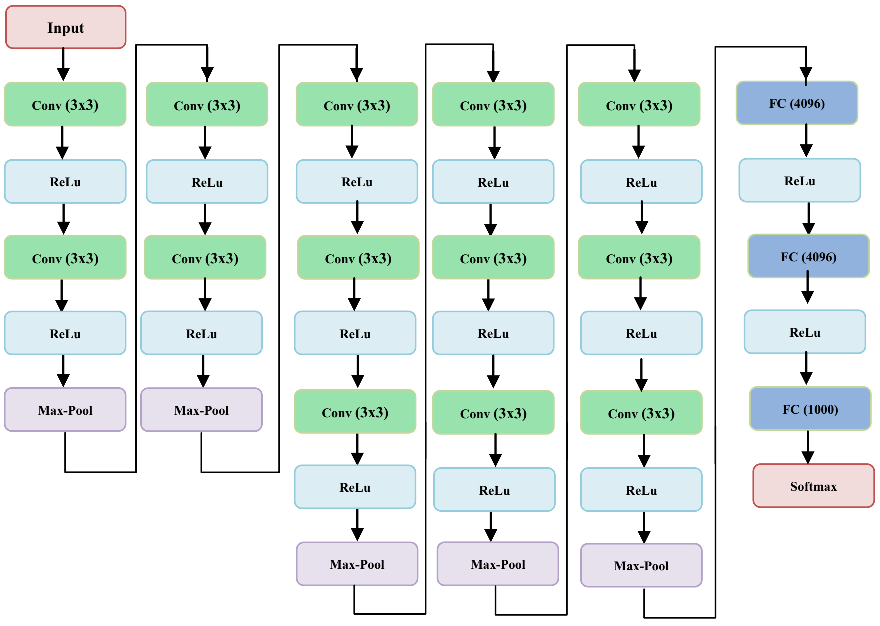


Fig. 4.5 Architecture diagram of VGGNet-16

4.6 GoogleNet

GoogleNet was introduced to address the following two issues:

(i) Increasing the depth of deep network implies increasing the number of parameters which makes the network more prone to overfitting. This is especially true if there are a limited number of labeled examples in the training dataset.
(ii) Increasing the network size uniformly increases the computational resources dramatically.

GoogleNet used an inception module as the basic building block of the network to address the above two issues. The inception module applies multiple convolutions, with different filter sizes, as well as pooling to the same input. It then concatenates the multiple-scale features produced by filters of different sizes.

GoogleNet uses 12 times fewer parameters than AlexNet. In this architecture, the use of average pooling at the end reduces the volume $7 \times 7 \times 1024$ to a volume $1 \times 1 \times 1024$ as a result of which it saves a huge number of parameters.

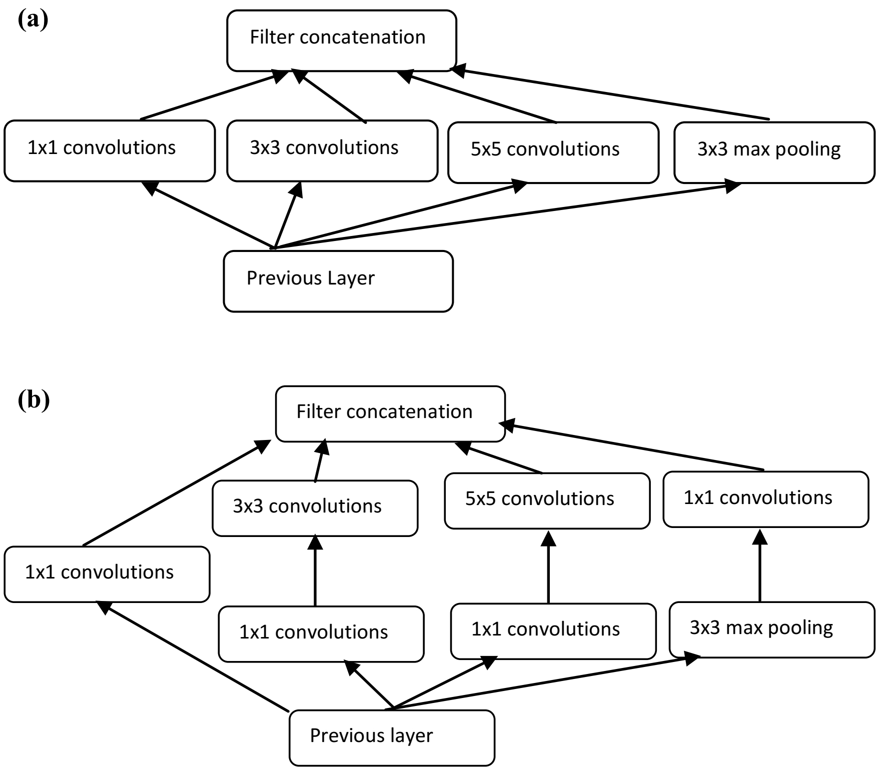


Fig. 4.6 Inception module of GoogleNet (**a**) naïve version, (**b**) with 1×1 convolution before computational expensive 5×5 and 3×3 convolutions

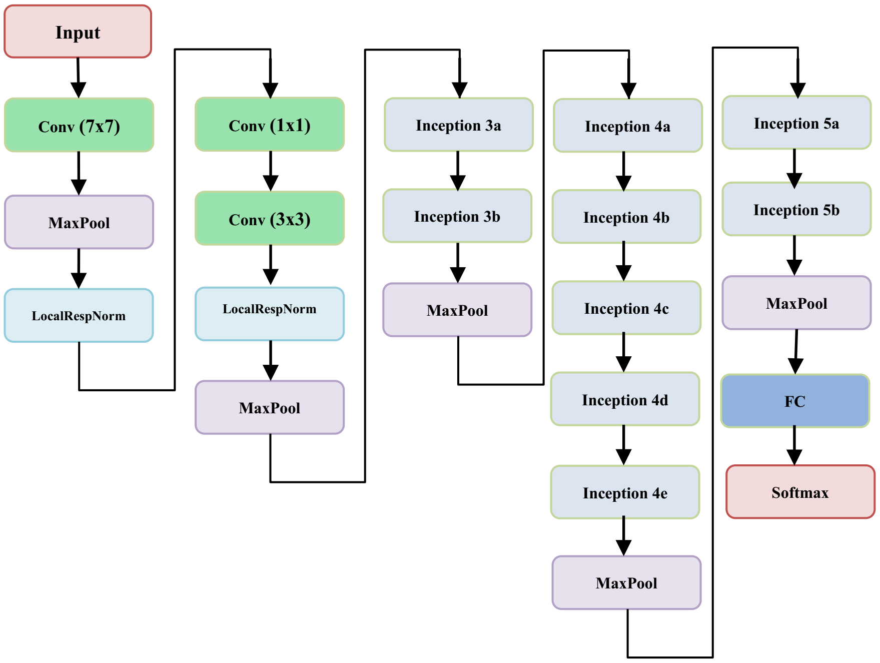


Fig. 4.7 Architecture diagram of GoogleNet

In order to evade patch alignment problems, the inception architecture is limited to filter sizes 5×5, 3×3 and 1×1. Each inception module of the network involves grouping of filter outputs which are concatenated into a single output vector which produces the input for the next module. As pooling operations are important for success of CNN networks, a substitute parallel pooling path has been added to each module. Figure 4.6a shows the inception module used in GoogleNet.

One problem with the inception modules is that even a 5×5 convolution can be computationally costly. This problem becomes worse when pooling units are added to the mix. The merging of the result of the pooling layer with the convolutional layers increases the number of outputs.

To address this problem, the GoogleNet architecture proposed second modification of applying dimension reductions wherever the computational requirements increased. This is achieved by using 1×1 convolution before making use of the computationally expensive 3×3 and 5×5 convolutions. The updated inception module based on this modification is shown in Fig. 4.6b.

A total of nine inception modules were used in GoogleNet, and each inception module had two layers. The network had a total of 27 layers (22 layers with parameters and 5 pooling layers). The 27 layers were made up of about 100 independent building blocks. The architecture diagram of GoogleNet is shown in Fig. 4.7, and details of various layers of GoogleNet are shown in Table 4.5.

Table 4.5 Details of various layers of GoogleNet

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Convolution	224 × 224	7 × 7	–	64	2	2	112 × 112	64
Max pool	112 × 112	–	3 × 3	–	2	0	56 × 56	64
Convolution	56 × 56	3 × 3	–	192	1	1	56 × 56	192
Max pool	56 × 56	–	3 × 3	192	2	0	28 × 28	192
Inception (3a)	28 × 28	–	–	–	–	–	28 × 28	256
Inception (3b)	28 × 28	–	–	–	–	–	28 × 28	480
Max pool	28 × 28	–	3 × 3	480	2	0	14 × 14	480
Inception (4a)	14 × 14	–	–	–	–	–	14 × 14	512
Inception (4b)	14 × 14	–	–	–	–	–	14 × 14	512
Inception (4c)	14 × 14	–	–	–	–	–	14 × 14	512
Inception (4d)	14 × 14	–	–	–	–	–	14 × 14	528
Inception (4e)	14 × 14	–	–	–	–	–	14 × 14	832
Max pool	14 × 14	–	3 × 3	–	2	0	7 × 7	832
Inception (5a)	7 × 7	–	–	–	–	–	7 × 7	832
Inception (5b)	7 × 7	–	–	–	–	–	7 × 7	1024
Avg-pool	7 × 7	–	7 × 7	–	–	0	1 × 1	1024
Dropout (40%)	–	–	–	1024	–	–	1 × 1	1024
Linear	–	–	–	1000	–	–	1 × 1	1000
Softmax	–	–	–	1000	–	–	1 × 1	1000

4.7 ResNet

As the number of layers of deep networks increases, its accuracy improves and the accuracy saturates once the network has converged. However, if the depth is further increased, then the performance starts getting degraded rapidly. This degradation is caused by adding more layers to an already converged deep model which results in higher training error. Thus, there is a need for a strategy that obtains an optimal deep network for a given application. ResNet was proposed with a residual learning framework that lets new layers to fit a residual mapping. It is easier to push the residual to zero when a model has converged than to fit the mapping by a stack of nonlinear layers.

Given an underlying mapping $H(x)$ to be fit by a few stacked layers, where x is the input to these layers, the residual learning uses the residual function $F(x) = H(x) - x$. It is easier to optimize the residual mapping than to optimize the original, and it can be realized by a feedforward neural network with shortcut connection as shown in Fig. 4.8. The shortcut link simply accomplishes identity mapping, and the output of

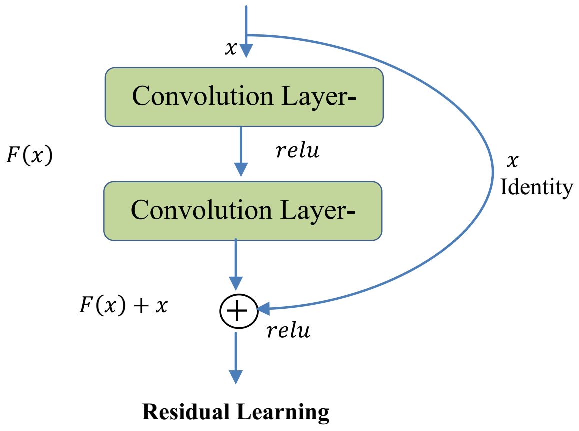


Fig. 4.8 ResNet residual learning building block

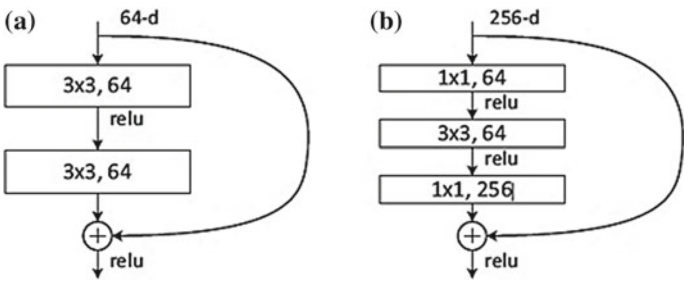


Fig. 4.9 **a** ResNet building block, **b** "Bottleneck" ResNet building block

the shortcut link is added to the outcomes of the stacked layers as shown in Fig. 4.8. The identity shortcut link does not add calculation complexity or parameters.

The residual function F uses a stack of 2 or 3 layers (more layers are also possible) as shown in Fig. 4.9. The building block is defined by Eq. (4.1) as

$$y = F(x, \{W_i\}) + x \tag{4.1}$$

where x and y represent the input and output vectors of layers considered. The function $F(x, \{W_i\})$ represents the residual mapping which is to be learnt. The linear projection W_s is performed by a shortcut link to match the dimensions as in Eq. (4.2):

$$y = F(x, \{W_i\}) + W_\mathrm{s}x \tag{4.2}$$

The architecture diagram of ResNet-34 is shown in Fig. 4.10, and Table 4.6 gives details of various layers of ResNet-34.

4.8 Densely Connected Convolutional Network (DenseNet)

The information present in the input data passes through many layers in a deep network. This information can vanish and wash out by the time it reaches the end. To address this problem, Densely Connected Convolutional Network (DenseNet) has been proposed that concatenates features from previous layers instead of combining the features from the previous layers.

The ResNet used residual connection where input to the ith layer was obtained by summation of outputs from the previous layers. In contrast, the DenseNet concatenates outputs from the previous layers. The DenseNet is a type of network in which each layer is connected to every other layer in a feedforward fashion. In this network, features are never combined by means of summation before they are passed into a layer, rather features are combined by the process of concatenation. Thus, the nth layer in the network has n number of inputs, comprising the feature maps of all the previous convolutional layers. Its own feature maps are passed on to all $(N - n)$ layers that follow it. This results in $\frac{N(N+1)}{2}$ connections in an N-layered network, instead of just N. As the network employs dense connections, it is referred as Dense Convolutional Network (DenseNet).

An image represented by x_0 is passed through a convolutional network comprising N layers; each of the layers implements a nonlinear transformation represented by $H_n(.)$, where n indexes the layer of the network. $H_n(.)$ can be a function representing multiple operations, for example, it can be a function of three operations: batch normalization, rectified linear units, and convolution (BN, ReLU, Conv(3×3)) as shown in Fig. 4.11. The output of the nth layer is denoted by x_n. Direct connections from any layer to all subsequent layers were introduced in the DenseNet. Thus, the

Table 4.6 Details of various layers of ResNet-34

Layer name	Input size	Filter size	Window size for pooling	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	224 × 224	7 × 7	–	64	2	2	112 × 112	64
Conv 2_x	112 × 112	–	3 × 3	–	2	0	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
	56 × 56	3 × 3	–	64	1	1	56 × 56	64
Conv 3_x	56 × 56	3 × 3	3 × 3	128	2	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
	28 × 28	3 × 3	–	128	1	1	28 × 28	128
Conv 4_x	28 × 28	3 × 3	3 × 3	256	2	0	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
	14 × 14	3 × 3	–	256	1	1	14 × 14	256
Conv 5_x	14 × 14	3 × 3	3 × 3	512	2	0	7 × 7	512
	7 × 7	3 × 3	–	512	1	1	7 × 7	512
	7 × 7	3 × 3	–	512	1	1	7 × 7	512
	7 × 7	3 × 3	–	512	1	1	7 × 7	512
	7 × 7	3 × 3	–	512	1	1	7 × 7	512
	7 × 7	3 × 3	–	512	1	1	7 × 7	512
Avg-pooling layer	7 × 7	–	7 × 7	–	–	–	1 × 1	1000
Fully connected layer	1000 Classes							

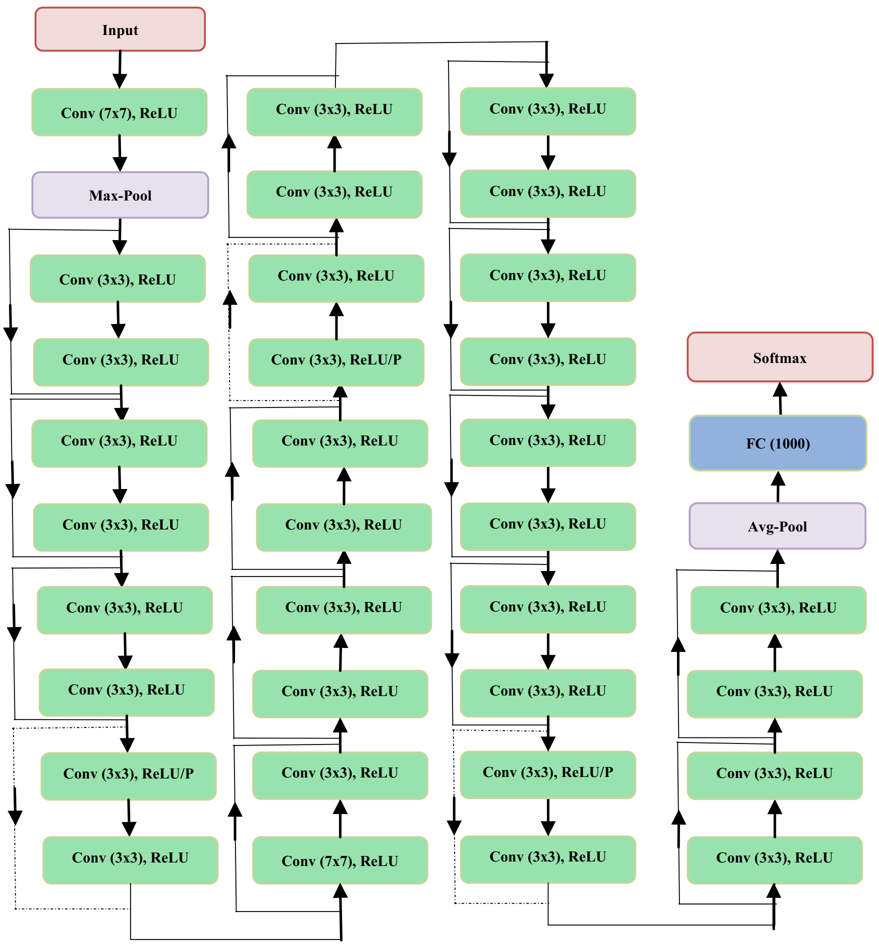


Fig. 4.10 Architecture diagram of ResNet-34 layers

nth layer receives input feature maps $x_0, \ldots, x_{n-1}$ of all the proceeding layers to produce output x_n as shown in Eq. (4.3):

$$x_n = H_n\left(\left[x_0, x_1, \ldots, x_{n-1}\right]\right) \tag{4.3}$$

where $\left[x_0, x_1, \ldots, x_{n-1}\right]$ represents the concatenation of feature maps formed in layers $0, \ldots, n-1$.

Figure 4.11 shows the dense block diagram along with a transition layer. The $H_n(.)$ function in each dense block represents three operations: batch normalization, rectified linear units, and convolution (BN, ReLU, Conv(3 $\times$ 3)). Each layer of DenseNet has typically many inputs but the layer produces only k output feature maps. The network uses 1 × 1 convolution as a bottleneck layer before each 3 × 3 convolution to reduce the number of input feature maps which improves the computational efficiency. The bottleneck version of $H_n(.)$ function represents the following operations: (BN, ReLU, Conv(1 × 1), BN, ReLU, Conv(3 × 3)). The transition layer between two dense blocks uses 1 × 1 convolution and 2 × 2 average pooling to further reduce the number of feature maps.

Architecture diagram of DenseNet is shown in Fig. 4.12. Details of various layers of DenseNet are given in Table 4.7.

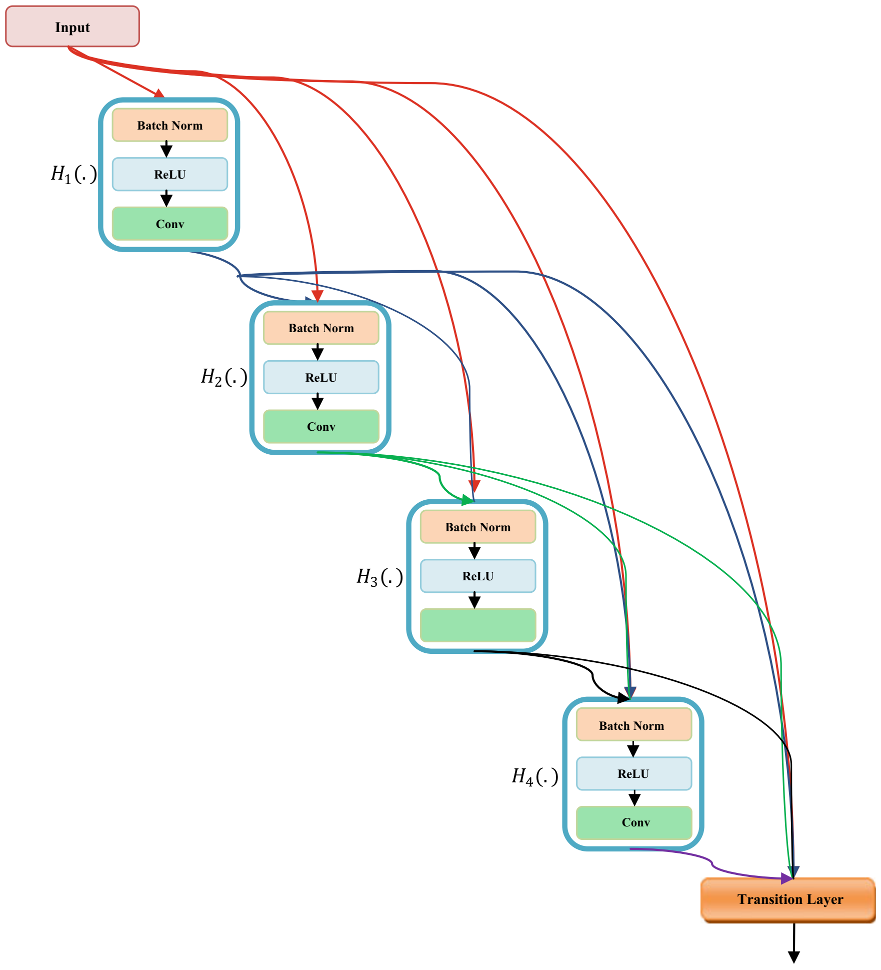


Fig. 4.11 A five-layer dense block, each layer takes all preceding feature maps as input

Table 4.7 Details of various layers of DenseNet-121

Layers		Input size	Filter size		Window size for pooling	Stride	Padding	Output size
Convolution		224 × 224	7 × 7		–	2	2	112 × 112
Max-pooling		112 × 112	–		3 × 3	2	0	56 × 56
Dense block (1)		56 × 56	1 × 1 3 × 3	×6	–	–	0	56 × 56
Transition layer (1)	Convolution	56 × 56	1 × 1		–		0	56 × 56
	Average pooling	56 × 56	–		2 × 2	2	0	28 × 28
Dense block (2)		28 × 28	1 × 1 3 × 3	×12	–		0	28 × 28
Transition layer (2)	Convolution	28 × 28	1 × 1		–		0	28 × 28
	Average pooling	28 × 28	–		2 × 2	2	0	14 × 14
Dense block (3)		14 × 14	1 × 1 3 × 3	×24	–		0	14 × 14
Transition layer (3)	Convolution	14 × 14	1 × 1		–		0	14 × 14
	Average pooling	14 × 14	–		2 × 2	2	0	7 × 7
Dense block (4)		7 × 7	1 × 1 3 × 3	×16	–	–	0	7 × 7
Average pooling		7 × 7	–		7 × 7	–	0	1 × 1
Fully connected layer		1000 Neurons						
Softmax		N Classes (N = 100, 1000)						

4.9 Capsule Network

The human brain has a mechanism to route image data to parts of the brain where it is perceived. Convolutional neural networks use layers of filters to extract high-level features from image data but the routing mechanism is absent in it.

Capsule Network (CapsNet) has been proposed that provides a routing mechanism. A CapsNet can have many capsule layers, where each layer is comprised of a number of capsules. A capsule is a group of neurons that can perform computations on their inputs and then compute an output in the form of a vector. The computations of the neurons within a capsule can represent various properties like pose, size, position, deformation, orientation, etc. of an entity (object or a part of an object) that is present in a given image. CapsNet uses the length of the output vector to represent the existence of an entity. The length of the output vector of a capsule is not allowed to exceed 1 by applying a nonlinearity that leaves the orientation of the vector unchanged but scales down its magnitude. A CapsNet proposed incorporating

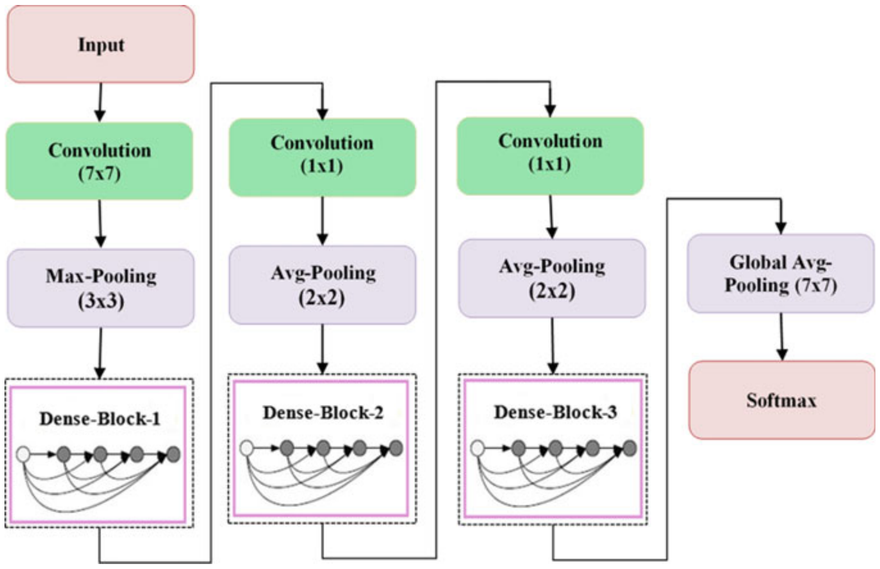


Fig. 4.12 Architecture diagram of DenseNet

a routing mechanism between two capsule layers. The routing mechanism makes a capsule in one layer to communicate to some or all capsules in the next layer.

Architecture diagram of a simple capsule network is shown in Fig. 4.13 and Table 4.8 gives details of various layers of the capsule network.

The first layer of this simple CapsNet is a convolutional layer that uses 256 filters of size of 9×9 with a stride of 1. It uses ReLU activation function. This layer converts pixel intensities into features which are used as inputs to the primary capsules.

The second layer of the CapsNet is the first capsule layer. This layer has 32 primary capsules. Each primary capsule has eight convolutional filters of size 9×9 used with a stride of two. Each primary capsule receives all 256 feature maps of size 20×20 produced by the first layer.

The primary capsules' layer produces 32 feature maps of size 6×6. Each feature map has a depth of 8, i.e., each feature map is an 8D vector.

The next layer is the second capsule layer which has one 16D capsule for each digit class. Each capsule in this layer receives input from all the capsules in the first capsule layer.

Table 4.8 Details of various layers of a simple capsule network

Layer name	Input size	Filter size	Pooling size for window	# Filters	# Capsules	Stride	Padding	Output size	# Feature maps
Conv 1	28 × 28	9 × 9	–	256	–	1	0	20 × 20	256
Capsule layer 1	20 × 20	9 × 9	–	8	32	2	0	6 × 6	32 (each with a depth 8)
Capsule layer 2	6 × 6	10-digit capsules, one for each digit							
Fully connected-1	512 Neurons								
Fully connected-2	1024 Neurons								
Fully connected-3	784 (which after reshaping gives back a 28 × 28 decoded image)								
Softmax	10 Classes								

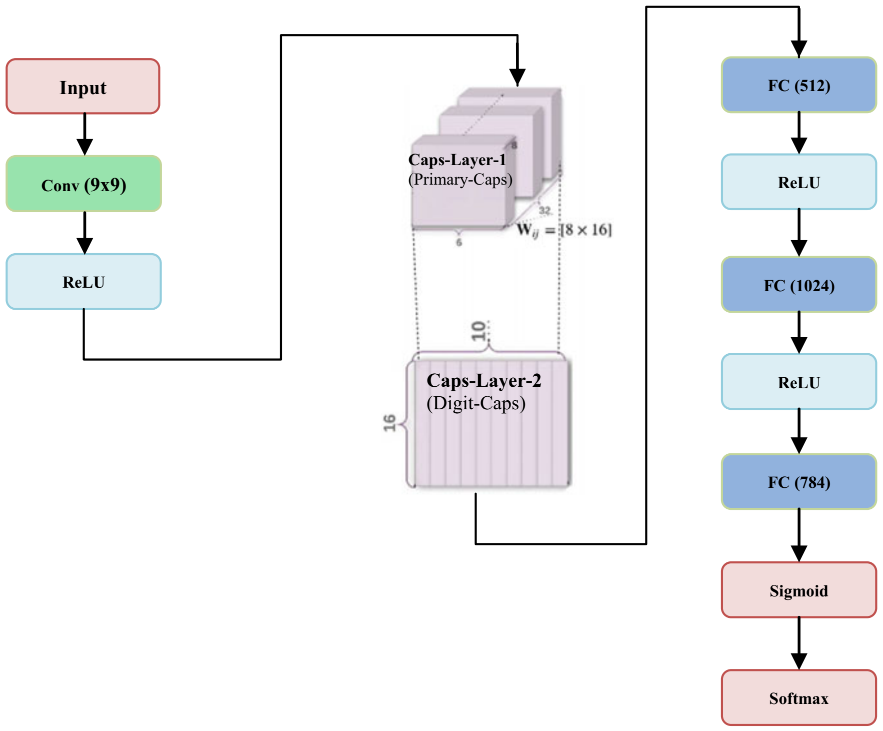


Fig. 4.13 Architecture diagram of a simple capsule network

The simple CapsNet has a routing mechanism between the two capsule layers only. Initially, a capsule output from the first capsule layer is sent to all capsules in the second capsule layer with equal probability. A dynamic routing mechanism is used to ensure that the output of a capsule is sent to the appropriate capsules in the next capsule layer and is determined by coupling coefficients. The coupling coefficients between a capsule in the first layer and all the capsules in the next layer sum to 1 and are determined by a routing softmax. The coupling coefficients can be learnt discriminatively at the same time as all other weights.

4.10 Challenges and Future Research Direction

Although Convolutional Neural Network (CNN)-based architectures have achieved a great success, there are still a number of areas that need to be investigated further. First, as the CNN-based architectures are composed of stack of layers or stack of modules, the challenge is to determine the optimal number of layers or optimal number of modules required for a given application. Determining the optimal internal structure of a module also needs further study.

Another challenge is to improve execution time. Pretraining of a module that is independent of other modules can result in parallel pretraining of various modules. A multicore high-performing Graphics Processing Units (GPUs) with parallel processing algorithms can be explored to speed up the execution time.

Datasets that are labeled manually involve vast expanses of human efforts. Unsupervised learning of convolutional neural networks can prove helpful here.

Choosing appropriate hyperparameters for a given application is a challenge. These hyperparameters have inner dependencies which make it expensive for fine-tuning. There is a scope to advance current optimization procedures for determining the optimal hyperparameters. This includes defining optimized modules and submodules that can be executed in parallel.

Bibliography

LeCun, Y.: LeNet-5, convolutional neural networks, p. 20. URL: http://yann.lecun.com/exdb/lenet (2015)

Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. (2012)

Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)

Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: European Conference on Computer Vision. Springer, Cham (2014)

Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2015)

He, K., et al.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2016)

Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: CVPR, vol. 1, no. 2, p. 3 (2017, July)

Gu, J., et al.: Recent advances in convolutional neural networks. Pattern Recogn. **77**, 354–377 (2018)

Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: Advances in Neural Information Processing Systems, pp. 3857–3867 (2017)

Chapter 5
Unsupervised Deep Learning Architectures

5.1 Introduction

The cascade of multiple layers of a deep learning architecture can be learnt in an unsupervised manner for the tasks like pattern analysis. A deep learning architecture can be trained one layer at a time, treating each layer in turn as an unsupervised restricted Boltzmann machine. Unsupervised deep learning algorithms are important because unlabeled data is more abundant than the labeled data. For applications with large volumes of unlabeled data, a two-step procedure is used: in the first step, a deep neural network is pretrained in an unsupervised manner; in the second step, a small portion of the unlabeled data is manually labeled, and then used for supervised fine-tuning of the deep neural network.

In this chapter, the following unsupervised deep learning networks are discussed: restricted Boltzmann machine, deep belief networks, autoencoders, and generative adversarial networks.

5.2 Restricted Boltzmann Machine (RBM)

Restricted Boltzmann Machine (RBM) is a generative model that can learn a probability distribution over its set of inputs. The word "restricted" means that connections between nodes of the same layer are prohibited. RBMs are used for training various layers in large networks one by one.

The training procedure of RBM involves adjusting the weights in such a manner so as to maximize the probability of generating the training data. RBM consists of two layers of neurons: visible layer for input vector $\boldsymbol{v}$ and hidden layer for the vector $\boldsymbol{h}$. All the neurons in the visible layer are connected to the neurons in the hidden layer, with no intralayer connections. Figure 5.1 depicts the architecture of RBM, with m visible units and n hidden units. The matrix W models the weights between

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_5

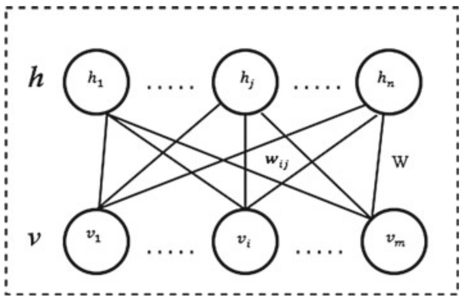


Fig. 5.1 Restricted Boltzmann machine

visible and hidden neurons; $\boldsymbol{w}_{ij}$ represents the weights between ith visible neuron and jth hidden neuron.

The probability distributions over couples ($\boldsymbol{v}$, $\boldsymbol{h}$) of visible units and hidden units in RBM are computed as follows:

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum_{\boldsymbol{v},\boldsymbol{h}} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}} \tag{5.1}$$

where the denominator is a normalization constant (partition function) that stands for the sum of $\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}$ overall possible configurations involving hidden and visible units. $E(\boldsymbol{v}, \boldsymbol{h})$ is the energy of a configuration ($\boldsymbol{v}$, $\boldsymbol{h}$) and is defined as

$$\begin{aligned} &E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i \boldsymbol{w}_{ij} h_j \\ &\text{or in matrix notation} \\ &E(\boldsymbol{v}, \boldsymbol{h}; W, a, b) = -a^T \boldsymbol{v} - b^T \boldsymbol{h} - \boldsymbol{v}^T W \boldsymbol{h} \end{aligned} \tag{5.2}$$

W represents weights; b is the bias for hidden units and a is the bias for visible units. The states of the visible vector $\boldsymbol{v}$ are associated to the input data and the hidden vector $\boldsymbol{h}$ presents the states of the hidden neurons, i.e., the hidden features. Given a data vector $\boldsymbol{v}$, the conditional probability that the units of the hidden layer will be activated is given as

$$p\big(h_j = 1|\boldsymbol{v}\big) = \sigma\left(b_j + \sum_{i=1}^{m} \boldsymbol{w}_{ij} v_i\right) \tag{5.3}$$

where $\sigma = \frac{1}{1+e^{-x}}$ is the sigmoid activation.

The hidden states can then be used for the reconstruction of the data by activating the units in visible layer with conditional probability:

$$p(v_i = 1|\boldsymbol{h}) = \sigma\left(a_i + \sum_{j=1}^{n} \boldsymbol{w}_{ij} h_j\right) \tag{5.4}$$

In order to reconstruct the input vector using Eq. (5.4), the hidden states are forced to be binary.

Contrastive divergence

RBMs are trained to improve the reconstruction ability, and thus to maximize the log-likelihood of training data over the training parameters for a given training example. Given a visible input vector, its probability overall the possible hidden vectors is computed as follows:

$$p(\boldsymbol{v}) = \frac{\sum_{\boldsymbol{h}} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum_{\boldsymbol{v},\boldsymbol{h}} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}} \tag{5.5}$$

The probability of a training vector can be increased by adjusting the network parameters (weights and biases) in order to lower the energy of that particular vector and raising the energy of all others that are hallucinated or sampled by the model. In order to adjust the weights and biases, we need to compute the derivative of the log probability with respect to network parameters $\theta \in \left\{a_i, b_j, \boldsymbol{w}_{ij}\right\}$ which is given by

$$\frac{\partial \log p(\boldsymbol{v})}{\partial \theta} = \underbrace{-\sum_{\boldsymbol{h}} p(\boldsymbol{h}|\boldsymbol{v}) \partial \frac{E(\boldsymbol{v},\boldsymbol{h})}{\partial \theta}}_{\text{Positive phase}} + \underbrace{\sum_{\boldsymbol{v},\boldsymbol{h}} p(\boldsymbol{v},\boldsymbol{h}) \partial \frac{\mathrm{E}(\boldsymbol{v},\boldsymbol{h})}{\partial \theta}}_{\text{Negative phase}} \tag{5.6}$$

We need a strategy to sample $p(\boldsymbol{h}|\boldsymbol{v})$ and another to sample $p(\boldsymbol{v}, \boldsymbol{h})$. In the positive phase, the visible layer is clamped on the input data and $\boldsymbol{h}$ is sampled from $\boldsymbol{v}$ while as in the negative phase, both $\boldsymbol{v}$ and $\boldsymbol{h}$ are to be sampled from the model. Computation of the first term is straightforward as the hidden nodes are independent of the visible nodes. Unfortunately, the second term is hard to compute. One possible strategy is to use a Markov chain Monte Carlo (MCMC) such as Alternating Gibbs Sampling (AGS). Each iteration of AGS consists of updating all the hidden units in parallel using Eq. (5.3) followed by updating all the visible units in parallel using Eq. (5.4), and then updating the hidden units once again using Eq. (5.3).

Using this procedure, Eq. (5.6) can be rewritten as

$$\frac{\partial \log p(\boldsymbol{v})}{\partial \theta} = \left\langle \partial \frac{E(\boldsymbol{v},\boldsymbol{h})}{\partial \theta} \right\rangle_0 + \left\langle \partial \frac{E(\boldsymbol{v},\boldsymbol{h})}{\partial \theta} \right\rangle_\infty \tag{5.7}$$

where $\langle . \rangle_0$ ($p_0 = p(\boldsymbol{h}|\boldsymbol{v}) = p(\boldsymbol{h}|\boldsymbol{x})$) and $\langle . \rangle_\infty$ ($p_\infty(\boldsymbol{v}, \boldsymbol{h}) = p(\boldsymbol{v}, \boldsymbol{h})$) correspond to the expectations under the distributions defined by data and model. However, this procedure is very time consuming, as the convergence obtained from this learning procedure is typically very slow. To solve this problem, a faster methodology—the Contrastive Divergence (CD) has been introduced, whereby $\langle . \rangle_\infty$ is replaced by $\langle . \rangle_k$ for small values of k. Basically, the idea is to initialize the visible units with a training sample, the hidden states can be inferred from Eq. (5.3) and in a similar manner, visible states can be inferred from hidden states using Eq. (5.4). This is equivalent to running Gibbs sampling using $k = 1$. This is illustrated in Fig. 5.2.

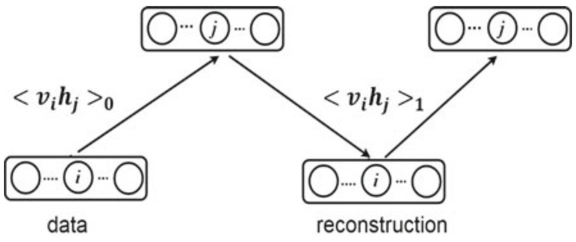


Fig. 5.2 Data and reconstruction in contrastive divergence training

The convergence of CD algorithm can be guaranteed if in every step of parameter updating the relationship, which the step number of Gibbs sampling and the learning rate must maintain is satisfied.

Accordingly changing Eq. (5.7), the update rules can be given as

$$\Delta \boldsymbol{w}_{ij} = \alpha\big(\langle v_i h_j\rangle_0 - \langle v_i h_j\rangle_1\big) \tag{5.8}$$

$$\Delta b_j = \alpha\big(\langle h_j\rangle_0 - \langle h_j\rangle_1\big) \tag{5.9}$$

$$\Delta a_i = \alpha(\langle v_i\rangle_0 - \langle v_i\rangle_1) \tag{5.10}$$

where α stands for the learning rate.

The adjustments are based on the difference between the first value in the chain $\langle v_i h_j\rangle_0$ and the last values $\langle v_i h_j\rangle_1$. The adjustment of weight $\boldsymbol{w}_{ij}$ depends only on the activations of units v_i and h_j. A simplified version of the same learning rule that uses the states of individual units instead of pairwise products is used for the biases.

Algorithm 5.1 The basic steps of CD algorithm are:

(i) Take a training sample $\boldsymbol{x}$, $\boldsymbol{v}^{(0)} \leftarrow \boldsymbol{x}$.
(ii) Compute the binary states of the hidden units $\boldsymbol{h}^{(0)}$ using Eq. (5.3).
(iii) Compute the reconstructed states of visible units $\boldsymbol{v}^{(1)}$ using Eq. (5.4).
(iv) Compute the binary states of hidden units using the reconstructed states of visible units obtained in step (iii) using Eq. (5.3).
(v) Update the weights, hidden and visible biases using Eqs. (5.8)–(5.10).

5.2.1 *Variants of Restricted Boltzmann Machine*

RBMs have been successfully applied in various applications such as character recognition, classification, face recognition, topic modeling, voice conversion, dimensionality reduction, musical genre categorization, natural language understanding, modeling of motion capture data, collaborative filtering, feature learning, and video sequences.

A number of variants of the RBMs have been proposed by researchers. These variants focus on different aspects of the model such as adding connection information between hidden and visible units. Some variants of RBM have lateral connections between the visible units—semi-restricted Boltzmann machines, while as others have directed connections between the visible units and between hidden units passing context information from the previous states to the current states—Temporal-Restricted Boltzmann Machines (TRBM). TRBM can be used to model complex time series sequence, wherein the decision made at each step requires some context information from the past.

One of the extensions of TRBM is recurrent Temporal-Restricted Boltzmann Machines (RTRBM), where each RBM uses a contextual hidden state received from the previous RBM to modulate its hidden unit bias. This type of RBM not only improves the prediction performance but it also identifies important dependency patterns in the data. Furthermore, a class of RTRBM referred to as structured RTRBM (SRTRBM), models the dependency structure using a graph. Conditional-restricted Boltzmann machine has been used to determine increments to the visible and hidden layer biases by adding a conditioning vector.

Fuzzy mathematics is used to expand the relation of hidden and visible units from constant to variable in a fuzzy-restricted Boltzmann machine. This replaces the model parameters by fuzzy numbers and conventional RBM energy function by fuzzy energy function. Conventional RBMs use binary visible and hidden units, and various extensions have been used to change the value type such as to use continuous units, units within the exponential family of softmax activated units, Poisson units, Student's t-distributed units, binomial units, Gaussian units with stochastic Gaussian noise, and rectified linear units.

Different variants of RBM's such as the mean-covariance RBMs, gated RBMs, spike-slab RBMs, factored three-way, and higher order models have also been used. In order to eliminate the influence of noise in the data, a Robust-Restricted Boltzmann Machine (RoBM) has been used that can achieve a better generalization by eliminating the influence of corrupted pixels. Compared to traditional algorithms, the RoBMs have shown impressive performance in the domain of visual recognition by accurately dealing with occlusions and noise. The effect of temperature parameter has also been considered on RBM—Temperature-based Restricted Boltzmann Machine (TRBM). In TRBMs, the temperature parameter can be used to control the firing neurons activity distribution by setting the slope parameter of the sigmoid function.

5.3 Deep Belief Network

Deep Belief Networks (DBN) are probabilistic graphical models made up of a hierarchy of stochastic latent variables using RBM as the basic building block. DBNs have also been shown to be universal approximators. It has been applied to various problems such as handwritten digit recognition, video and motion sequence recognition, dimension reduction, and data indexing.

DBN is a type of deep neural network composed of multiple layers, each layer consisting of visible neurons representing the layer input, and hidden neurons representing the layer output. The visible neurons are owned by the preceding layer, for which these neurons are hidden. The visible neurons are fully interconnected with the hidden ones. The distinctive feature of a DBN is that there are no connections between the visible neurons, and no connections between the hidden neurons. The connections are symmetric, and are exclusively between the visible neurons and the hidden ones. Figure 5.3 shows an example of DBN.

Just like RBMs, DBNs learn to reproduce the probability distribution of their inputs, without supervision. However, they are much better at it, for the same reason that deep neural networks are more powerful than shallow ones: real-world data is often organized in hierarchical patterns, and DBNs take advantage of that. Their lower layers learn low-level features in the input data, while higher layers learn high-level features. Just like RBMs, DBNs are fundamentally trained in unsupervised manner.

The training of DBNs involves two phases—unsupervised pretraining phase, which is performed in a bottom-up manner and a supervised fine-tuning phase. The unsupervised pretraining phase provides better initialization values to the weights than can be achieved by using the random initialization. The supervised fine-tuning is used to adjust the whole network. DBNs avoid the overfitting and underfitting problems, this is due to the fact that most of the network is trained in an unsupervised fashion which is data driven rather than label driven.

For unsupervised pretraining, the parameters for every consecutive pair of representational layers (Fig. 5.3) are learnt as an RBM. First, the bottom-most RBM is trained using raw training data as its visible layer. Once this RBM is trained, its latent activations are used as inputs to next RBM to obtain an encoded representation of the training data. Essentially, the hidden units of the current RBM are used as inputs to the next RBM. This procedure is repeated until a desired number of RBMs are obtained. Each RBM defines a layer of DBN. The stacking of RBMs results in

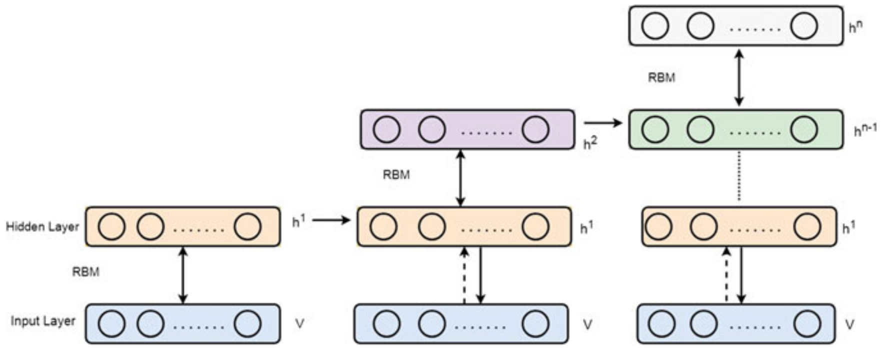


Fig. 5.3 A deep belief network

incremental feature discovery, where each RBM captures higher level correlations of the layers below it. When the topmost RBM is trained, a fine-tuning step is usually followed. This can either be done in an unsupervised manner using gradient descent on an estimate of the DBNs log-likelihood, or in a supervised manner if the DBN is used for classification or regression.

5.3.1 Variants of Deep Belief Network

Deep belief networks have achieved state-of-the-art results in different domains and are quite popular due to its ability to learn from large unlabeled datasets as a consequence of which many variants of DBN have been put forward. A sparse variant of deep belief network that uses sparse RBMs has been used for modeling higher order features. Another version of sparse deep belief network that trains the first level of the deep network with differentiable sparse coding, and then use sparse codes as input to train higher layers with the standard binary RBM has been used. A variant of deep belief network that uses a different type of top-level model has been introduced and its performance has been evaluated on a 3D object recognition task. The top-level model is a third-order Boltzmann machine that is trained using a hybrid algorithm that combines both generative and discriminative gradients. To improve the robustness of DBN to variations such as occlusion and random noise, sparsification and denoising algorithm has been put forward to make it more robust. To avoid catastrophic forgetting when the input distribution changes temporarily, M-DBN is used as an unsupervised modular DBN that prevents the forgetting of learned features in continual learning scenario. M-DBNs are composed of number of modules, and only those modules that best reconstruct a sample are trained. Additionally, M-DBN uses batch-wise learning scheme that adjusts each module's learning rate in proportion to the fraction of best reconstructed samples. M-DBN retains its effectiveness even when the input distribution changes, this is contrary to monolithic DBNs, that gradually forget the representations learnt before. Combinatorial deep belief networks have been used where one deep belief network is used for extracting features from the motion, while as the other network is used to extract features from the images. The output obtained from both the networks is then used as input to convolutional neural network, which then classifies the output into one of the actions.

To learn features from multiscale representation of images Multi-resolution Deep Belief Network (MrDBN) have been used. MrDBN involves constructing the Laplacian Pyramid for each image first, and then training DBN separately at each level of pyramid. A top-level RBM is then used to combine these DBNs into a single network, which is referred to as MrDBN.

DBN has also been used in classifying images by employing flexible Convolutional Deep Belief Network (CDBN). CDBN is a generative model that has shown good performance in many visual recognition tasks.

5.4 Autoencoders

Autoencoders (AEs) also called as autoassociators are capable of learning efficient representations of the input data called codings, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction. More importantly, autoencoders act as powerful feature detectors, and can be used for unsupervised pretraining of deep neural networks. Autoencoders are also capable of generating new data that looks very similar to the training data and can serve as a generative model. For example, one can train an autoencoder on pictures of faces, and it can then be used to generate new faces. A Multilayer Perceptron (MLP) in auto-association mode can achieve dimensionality reduction and data compression. One of the main tasks of the autoencoder is to find out a representation that can be used to reconstruct the input data with high accuracy. The general process of autoencoder is shown in Fig. 5.4. The goal of the training process is to transform the input vector into a low-dimensional coded vector and to reconstruct the input data from the corresponding code with minimum reconstruction error. The coding process basically involves learning features from the input data. For each input vector, AE extracts useful features and filters the unwanted information.

The difference between MLP and AE is that an MLP is trained to predict a target value Y given input X, while as AE is used to reconstruct the input. The AE converts the input X into a code h during the encoding process with the help of weight matrix W. It reconstructs $\widetilde{X}$ during the decoding process from h by using weight matrix W' (the decoder weight matrix is the transpose of encoder weight matrix W). During the process of training AEs, parameter optimization is followed so as to minimize the error between the input vector X and the reconstruction $\widetilde{X}$.

Generally, if the dimension of the internal layer is less than that of the input layer, autoencoder performs dimensionality reduction task. On the contrary, if the hidden layer size is greater, then we enter the realm of feature detection.

Considering an unlabelled training set $\{x_1, x_2, \ldots\}$, where $x_i \in \mathbb{R}^n$, the AE can be considered as an unsupervised learning algorithm, where the target vector $h_{(W,b)}$ is set equal to the input vector x (i.e., $h_{(W,b)}(x_i) = x_i$). As the input vector is taken as the target vector, an AE obtains a reconstruction of the input vector. The basic structure of an autoencoder is shown in Fig. 5.5. The first part of the AE that converts the input vector to an internal representation is called encoder (or the recognition network):

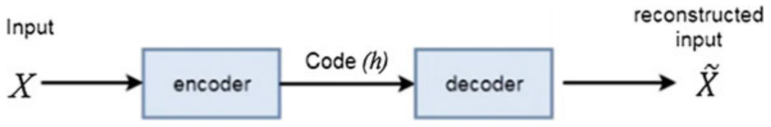


Fig. 5.4 The general process of an autoencoder

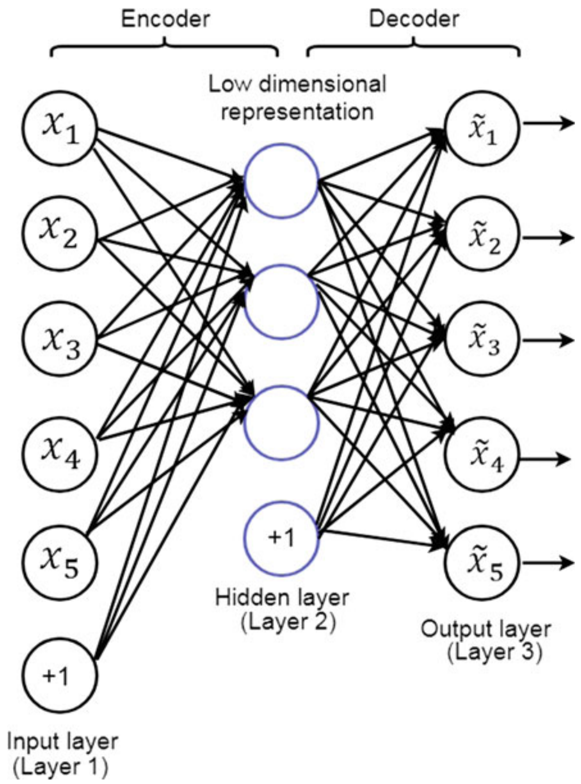


Fig. 5.5 Basic structure of an autoencoder

$$a^{(2)} = f\left(W^{(1)}x + b^{(1)}\right) \tag{5.11}$$

where $f(.)$ is the activation function of the encoder. The second part of the AE that converts the internal representation into the output vector is called decoder (or generative network):

$$h_{(W,b)}(x) = g\left(W^{(2)}a^{(2)} + b^{(2)}\right) \tag{5.12}$$

where $g(.)$ is the activation function of the decoder. The learning process is simply minimizing a loss function

$$L(x, g(f(x))) \tag{5.13}$$

where L is a loss function such as the mean squared error, penalizing $g(f(x))$ for being dissimilar from x.

An autoencoder typically has the same architecture as a multilayer perceptron except that the number of neurons in the output layer must be equal to the number of inputs. The example shown below has just one hidden layer composed of three neurons (the encoder) and one output layer composed of five neurons (the decoder). The outputs are also called as reconstructions since the autoencoder tries to reconstruct the inputs, and the cost function contains a reconstruction loss that penalizes the model when the reconstructions are different from the inputs. The most commonly used activation functions for encoder and decoder are sigmoid, identity (linear) function ($g(x) = x$) or the hyperbolic tangent function. A nonlinear activation function for the encoder and a linear activation function for the decoder is the obvious choice when the input values are not constrained to lie in the range of [0, 1] or [−1, 1]. This autoencoder with linear decoder can produce unbounded output with values greater than 1 or less than 0.

The backpropagation algorithm is the commonly used algorithm employed in training of autoencoders in order to find the appropriate value of model parameters $\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$ for reconstruction of the original input vector. Autoencoders can be forced to learn useful representations of the data by imposing some constraints on it. Such constraints can include making the hidden layer to have a small number of hidden nodes, in which case the network learns a compressed representation of input data. For example, if 30-by-30 image is used as an input, where $x_i \in \mathbb{R}^{900}$ and the hidden layer is confined to have only 50 units, the network learns a compressed representation of the input data. Alternative ways of constraining the autoencoder involve using greater number of hidden units than the input dimensions, such autoencoders are also known as regularized autoencoders.

5.4.1 *Variations of Auto Encoders*

There are various variants of autoencoders proposed by researchers. Table 5.1 lists some well-known flavors of autoencoders and it briefly summarizes their advantages and characteristics.

5.4.1.1 Denoising Autoencoders

The denoising autoencoder is a basic autoencoder with only one major difference. In denoising autoencoder, the input is first partially corrupted and then fed to the network. The network is trained such that the original input is reconstructed from the partially corrupted data. The main rationale behind using this criterion is that it forces the autoencoder to learn the main underlying structure in the input data which may be sufficient to appropriately reconstruct the original input vector.

Traditionally, autoencoders minimize loss function L that penalizes $g(f(x))$ for being dissimilar from x

$$L(x, g(f(x))) \tag{5.14}$$

Table 5.1 Variants of autoencoders

Autoencoder	Characteristics	Advantages
Sparse autoencoders	The representation is forced to be sparse by adding a sparsity penalty	The network makes the categories more separable and meaningful, which improves the performance of the network
Denoising autoencoders	The network can reconstruct the correct input from the corrupted data	The network is robust to noise
Contractive autoencoder	The reconstruction error function is augmented with an analytic contractive penalty	Good at capturing the local directions of variation determined by the data
Convolutional autoencoder	All locations in the input share weights	Allows to use 2D image structure
Zero bias autoencoder	Autoencoder is trained by using an appropriate shrinkage function without additional regularization	Achieves better results on data with high intrinsic dimensionality

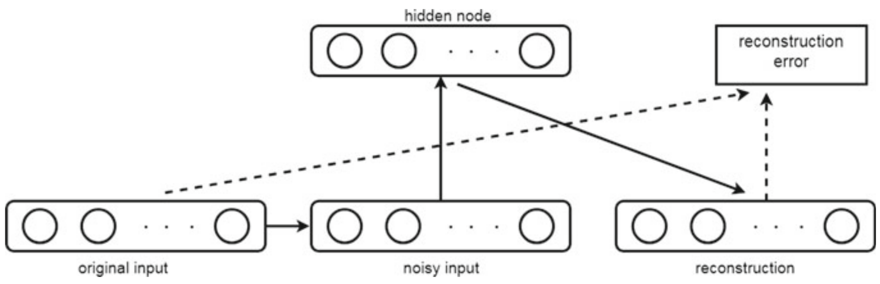


Fig. 5.6 Denoising autoencoder

A denoising autoencoder (DAE) instead minimizes:

$$L\left(x, g\left(f\left(\hat{x}\right)\right)\right), \tag{5.15}$$

where $\hat{x}$ is a copy of x that has been corrupted by some form of noise. Denoising autoencoders must, therefore, undo this corruption rather than simply copying their input. The process of a DAE is shown in Fig. 5.6.

A DAE can extract the noise-free version of the input data. The statistical dependencies inherent in the input data can be exploited to minimize the adverse effects of the noisy input data corrupted in a stochastic manner. If the type and level of the corrupting noise can be determined, then the implementation of the DAE is easier.

5.4.1.2 Contractive Autoencoders

Contractive Autoencoder (CAE), followed after the DAE, shares the motivation of learning robust representations. A DAE injects noise in the training data to make the mapping robust, a CAE adds an analytic contractive penalty to the standard cost function in the reconstruction stage to achieve robustness. The penalty term is added to correct the sensitivity of the features with respect to the inputs.

It has been found that the addition of penalty term results in more robust features that are insensitive to small input changes. Besides, the penalty can be used to address the trade-off between the reconstruction accuracy and robustness. CAEs produce results that are identical to or even better than those obtained by other regularized AEs such as DAEs. A DAE with small corruption noise can be considered as a type of CAE where the contractive penalty is on both the encoder as well as the decoder rather than just on the encoder as is the case with CAEs. CAEs are generally used in feature engineering as only the encoder portion is used for feature extraction.

5.5 Deep Autoencoders

Deep Autoencoders are auto-associative networks with more than one hidden layer. Generally, AE's with single layer are not able to extract features that are discriminative and representative of raw data. Therefore, the concept of deep autoencoders or stacked autoencoders has been put forward. Adding more layers helps the autoencoder learn more complex codings. However, one must be careful not to make the autoencoder too specialized. An encoder specializes if it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously, such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful generic data representations in the process and it is unlikely to generalize well to new instances.

The architecture of the stacked autoencoder is typically symmetrical with regards to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for MNIST (handwritten digit recognition dataset) may have 784 inputs, followed by a hidden layer with 300 neurons, then a central hidden layer of 150 neurons, then another hidden layer with 300 neurons and an output layer with 784 neurons. This stacked autoencoder is represented in Fig. 5.7.

The deep stacked autoencoder can be implemented much like a regular deep MLP except that there are no labels. The deep autoencoder network is formed by a series of autoencoder networks, stacked one above another in a feature hierarchy. Each autoencoder aims to minimize the reconstruction error of the previous layer.

Deep stacked autoencoders are generally trained using greedy layer-wise unsupervised learning followed by supervised fine-tuning. The main idea of the unsupervised pretraining is to provide a good initialization value of the weights of the network before a supervised learning algorithm is applied for fine-tuning. Besides, unsupervised pretraining also results in better models as it relies mainly on unla-

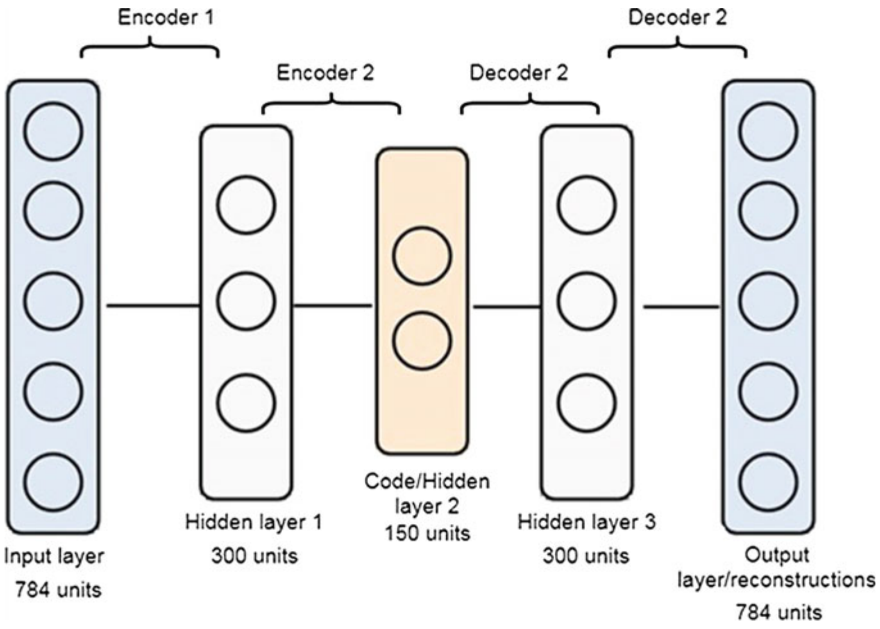


Fig. 5.7 Deep stacked autoencoder

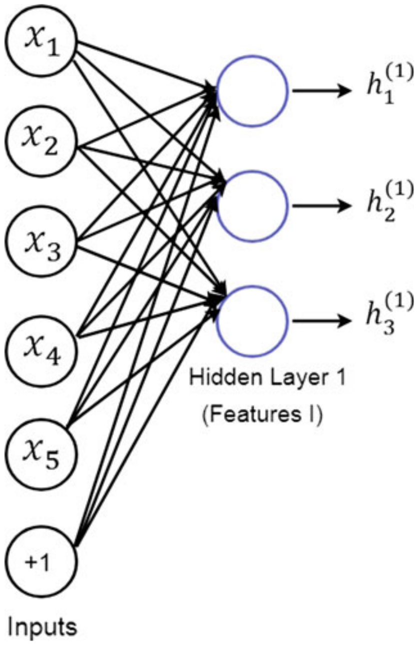


Fig. 5.8 Training first hidden layer of autoencoder

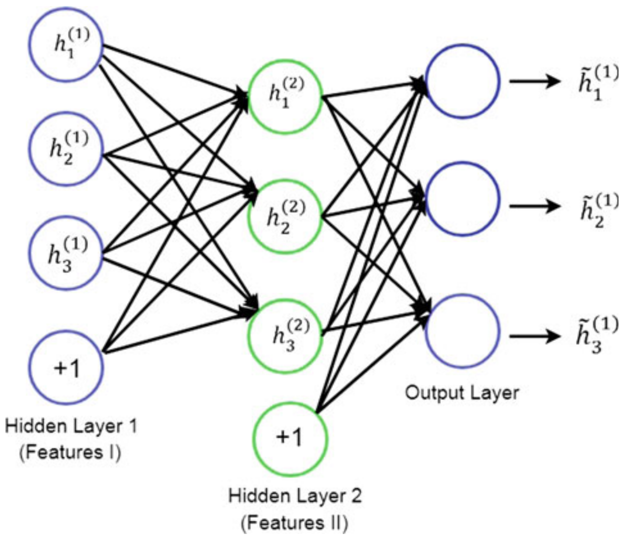


Fig. 5.9 Training second hidden layer on the features obtained from the first autoencoder

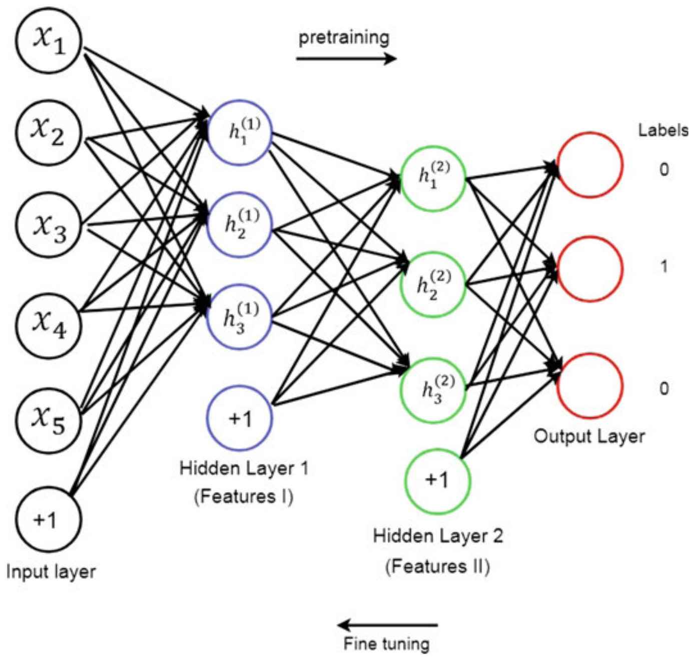


Fig. 5.10 Fine-tuning of the network after adding an output layer to the stack of pretrained hidden layers

belled data. The subsequent supervised fine-tuning involves globally adjusting the weights learned using unsupervised pretraining. The autoencoder shown in Fig. 5.5 are first trained using backpropagation algorithm (using gradient descent optimization) without using labels as discussed in the previous section. After this, the last layer of this network (the decoder) is dropped, while as the hidden layer with its parameters $\{W^{(1)}, b^{(1)}\}$ (the encoder) is retained as shown in Fig. 5.8.

The second autoencoder is trained with the features obtained from the first autoencoder as shown in Fig. 5.9. The parameters of the first autoencoder are kept unchanged while training the second autoencoder. This way, the network is trained one layer at a time, i.e., each layer is trained greedily and the weights so obtained are used as initial weights for the final fine-tuning process.

Thus, the first autoencoder is trained on the input data x_i using the backpropagation algorithm to obtain the features at the first level $h^{(1)(i)}$. The features obtained at the first level are used as inputs for training the next autoencoder. The second autoencoder is trained in a manner similar to the first autoencoder to obtain the second layer of representations $h^{(2)(i)}$. In this manner, each autoencoder is trained by using the representations learnt by the previous autoencoder. Only the parameters of the autoencoder that is currently being trained are updated, while as the parameters of previous autoencoders are kept fixed. Finally, an output layer is added to this stack of trained autoencoders and the whole network is trained by a supervised learning algorithm (using labeled data). In Fig. 5.10, two autoencoders are pretrained and an output layer is then added to it to form the final network.

5.6 Generative Adversarial Networks

Generative Adversarial Networks (GANs), are the models that learn any distribution of the data and primarily focusses on generating samples from the learnt distribution. They enable generation of the reasonably realistic worlds that are indistinguishable to our own in any domain: audio, images, speech. A GAN is made of two main components: Generator and Discriminative, which are in constant battle with each other throughout the training process.

- Generator network—A generator $G(z)$ takes as input a random noise and tries to generate a sample of data. Figure 5.11 shows generator $G(z)$ takes an input z from the probability distribution $p(z)$ and generates data that is then fed into a discriminator network $D(x)$.
- Discriminator network (or adversary)—the discriminator network $D(x)$ takes input either from the real data or from the network generated data and tries to predict whether the input is real or generated. It takes an input x from the real data distribution $p_{\text{data}}(x)$ and then solves a binary classification problem giving output in the range from 0 to 1. Figure 5.11 gives the basic architecture of GAN.

Basically, the task of the generator is to generate natural-looking images and the task of the discriminator is to decide whether the image is generated or real.

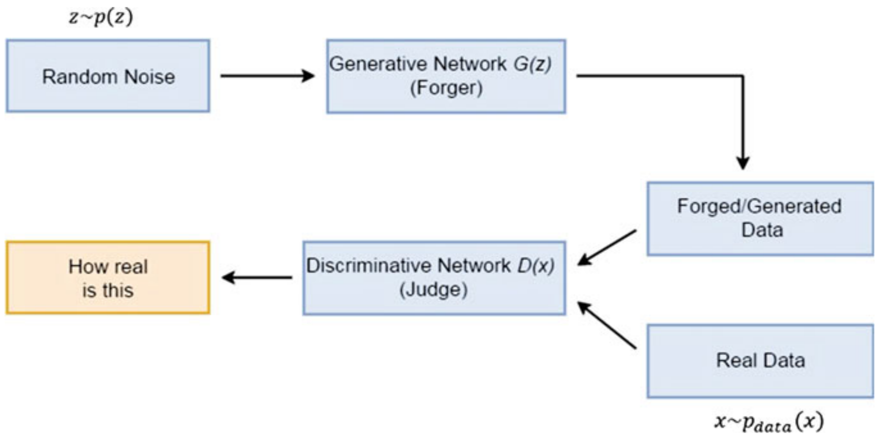


Fig. 5.11 Architecture of GAN

This can be thought of as a minimax two-player game where the performance of both the networks improves over time. In this game, the generator tries to fool the discriminator by generating real images as far as possible and the generator tries to not get fooled by the discriminator by improving its discriminative capability.

The generator network is trained to generate new data instances that look indistinguishable from the real ones as the training goes on, to the degree that it becomes difficult for a discriminator network to predict which is authentic and which is fake. The discriminator network is trained to gradually improve its capabilities of identifying fake data by fixing a higher level of realism for the generated data.

The objective function of GAN is defined as

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}\big[\log D(x)\big] + E_{x \sim p_z(z)}\big[\log(1 - D(G(z)))\big] \quad (5.16)$$

At the equilibrium point, which is the optimal point in the minimax game, the first network will model the real data and the second network will output a probability of 0.5 as the output of the first network real data. Sometimes, the two networks eventually reach equilibrium, but this is not always guaranteed and the two networks can continue learning for a long time.

GAN is a network where optimization process is not seeking to minimize a fixed training criteria but it seeks to establish an equilibrium between two forces. It is a dynamic process as the two forces change during every step of the optimization process. For this reason, GANs are known to be difficult to train, requiring a careful tuning of training parameters and model architecture.

The fact that the amount of available unlabeled data is much larger than the amount of labeled data, it has made GANs popular because of their ability to tackle the important challenge of unsupervised learning. Another reason for their popularity is that GANs are able to generate the most realistic images among the generative models.

5.7 Challenges and Future Research Direction

Although unsupervised learning algorithms had a catalytic effect in reviving the interest in deep learning, but much research needs to be done to improve the unsupervised deep learning algorithms. In particular, unsupervised learning algorithms are not good at disentangling the underlying factors that account for how the learning data is spread in the hyperspace. By making unsupervised learning algorithms better at disentangling the underlying factors that account for data variations in the hyperspace, the information can be used for efficient classification and for efficient transfer learning.

Advancing unsupervised learning by exploiting new sources of unlabeled data and mapping relationships from input to output and vice versa needs to be explored. Exploiting the relationship from output to input is closely connected to building conditional generative models. To this end, generative adversarial networks are a promising direction where the long-standing concept of analysis by synthesis in pattern recognition and machine learning is likely to return to the spotlight in the near future in solving different tasks.

Bibliography

Afzal, S., Wani, M.A.: Improving performance of deep networks on handwritten digit classification. In: 2017 4th International Conference on Computing for Sustainable Global Development (INDIACOM), pp. 4238–4241. IEEE (2017)

Afzal, S., Wani, M.A.: Deep neural network architectures: a review. In: 2018 5th International Conference on Computing for Sustainable Global Development (INDIACOM), pp. 3024–3030. IEEE (2018)

Afzal, S., Wani, M.A.: Training and model structure of deep architectures. Artif. Intell. Syst. Mach. Learn. **10**(2), 38–46 (2018)

Bengio, Y.: Learning deep architectures for AI. Found. Trends® Mach. Learn. **2**(1), 1–127 (2009)

Bengio, Y., Courville, A., Vincent, P.: Representation learning: a review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1798–1828 (2013)

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In Advances in neural information processing systems, pp. 153–160 (2007)

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning?. J. Mach. Learn. Res. **11**(Feb), 625–660 (2010)

Goroshin, R., LeCun, Y.: Saturating auto-encoders. arXiv preprint arXiv:1301.3577 (2013)

Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Comput. **14**(8), 1771–1800 (2002)

Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786), 504–507 (2006)

Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)

Larochelle, H., Bengio, Y., Louradour, J., Lamblin, P.: Exploring strategies for training deep neural networks. J. Mach. Learn. Res. **10**(Jan), 1–40 (2009)

Lee, H., Ekanadham, C., Ng, A.Y.: Sparse deep belief net model for visual area V2. In: Advances in Neural Information Processing Systems, pp. 873–880 (2008)

Poultney, C., Chopra, S., Cun, Y.L.: Efficient learning of sparse representations with an energy-based model. In: Advances in Neural Information Processing Systems, pp. 1137–1144 (2007)

Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y.: Contractive auto-encoders: explicit invariance during feature extraction. In: Proceedings of the 28th International Conference on International Conference on Machine Learning, pp. 833–840. Omnipress (2011)

Tieleman, T., Hinton, G.: Using fast weights to improve persistent contrastive divergence. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 1033–1040. ACM (2009)

Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P. A. Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th international conference on Machine learning, pp. 1096–1103. ACM (2008)

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**(Dec), 3371–3408 (2010)

Wani, M.A., Afzal, S.: Gain parameter and dropout-based fine tuning of deep networks. Int. J. Intell. Inf. Database Syst. **11**(4), 236–254 (2018a)

Wani, M.A., Afzal, S.: Optimization of deep network models through fine tuning. Int. J. Intell. Comput. Cybern. **11**(3), 386–403 (2018b)

Chapter 6
Supervised Deep Learning in Face Recognition

6.1 Introduction

The three main challenging problems in face recognition, i.e., recognizing a face with different expressions, recognizing a face under different lighting conditions, and recognizing a face in different poses are considered here. In recent years, Convolutional Neural Network (CNN) has been widely applied to face recognition problem because of its good performance. The application of CNN architectures for the above three challenging problems is discussed in this chapter.

6.2 Deep Learning Architectures for Face Recognition

The power of a Convolutional Neural Network (CNN) lies in extracting a set of discriminating feature maps at several levels of abstraction. The success of a CNN architecture is ascribed to its ability to learn rich image features. Two deep learning architectures based on a Convolutional Neural Networks are discussed in this chapter for face recognition: VGG-face architecture and modified VGG-face architecture.

6.2.1 VGG-Face Architecture

The VGG-face architecture used here for face recognition consisted of 13 convolutional layers, 5 pooling layers, and 3 fully connected layers. The architecture diagram of VGG-face architecture is given below in Fig. 6.1.

Table 6.1 shows the details of 13 convolutional layers, 5 pooling layers, 3 fully connected layers, and the softmax layer of the VGG-face architecture. The network uses filters of size 3×3 in all 13 convolutional layers. All the convolution layers are followed by the Rectified Linear Unit (ReLU) activation function. The network uses

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_6

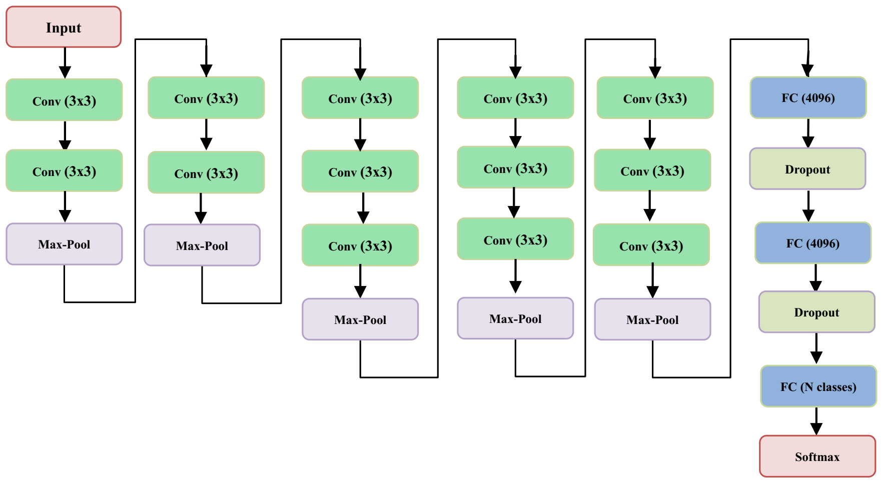


Fig. 6.1 VGG-face architecture for face recognition

max-pooling in all the five pooling layers with a stride of two. The last three layers are Fully Connected (FC).

6.2.2 *Modified VGG-Face Architecture*

The modified VGG-face architecture used for face recognition consisted of five convolutional layers, three pooling layers, and three fully connected layers. Figure 6.2 shows the architecture of modified VGG-face architecture for face recognition.

Table 6.2 shows the details of five convolutional layers, three pooling layers, three fully connected layers, and the Softmax layer of the modified VGG-face architecture. The first convolutional layer of the network uses filter of size 11 $\times$ 11 with a stride of four. The second convolutional layer uses filter of size 5 $\times$ 5 with a stride of one, and the last three convolutional layers use filters of size 3 $\times$ 3 with a stride of one. All the convolution layers are followed by the Rectified Linear Unit (ReLU) activation function. Normalization is performed after the first two activation functions. The network uses max-pooling in all the three pooling layers with a stride of two. The last three layers are Fully Connected (FC).

Face images are convolved with a filter of size 11 $\times$ 11. A large filter size is used when more pixels are necessary for the network to recognize an object. This layer uses 64 filters with a stride of 4 to reduce the size of the output feature maps to 54 $\times$ 54. The 64 feature maps are then normalized. In order to reduce the size of the normalized feature maps, max-pooling with stride of 2 is performed, which reduces feature maps to size 27 $\times$ 27.

Table 6.1 Details of VGG-face architecture for face recognition

Layer name	Input size	Filter size	# Filters	Window size for pooling	Stride	Padding	Output size	# Feature maps
Conv1	224 × 224	3 × 3	64	–	1	1	224 × 224	64
Conv2	224 × 224	3 × 3	64	–	1	1	224 × 224	64
Pool1	224 × 224	–	–	2 × 2	2	0	112 × 112	
Conv3	112 × 112	3 × 3	128	–	1	1	112 × 112	128
Conv4	112 × 112	3 × 3	128	–	1	1	112 × 112	128
Pool2	112 × 112	–	–	2 × 2	2	0	56 × 56	
Conv5	56 × 56	3 × 3	256	–	1	1	56 × 56	256
Conv6	56 × 56	3 × 3	256	–	1	1	56 × 56	256
Conv7	56 × 56	3 × 3	256	–	1	1	56 × 56	256
Pool3	56 × 56	–	–	2 × 2	2	0	28 × 28	
Conv8	28 × 28	3 × 3	512	–	1	1	28 × 28	512
Conv9	28 × 28	3 × 3	512	–	1	1	28 × 28	512
Conv10	28 × 28	3 × 3	512	–	1	1	28 × 28	512
Pool4	–	–	–	2 × 2	2	0	14 × 14	
Conv11	14 × 14	3 × 3	512	–	1	1	14 × 14	512
Conv12	14 × 14	3 × 3	512	–	1	1	14 × 14	512
Conv13	14 × 14	3 × 3	512	–	1	1	14 × 14	512
Pool5	–	–	–	2 × 2	2	0	7 × 7	
Fully connected 1	4096-neurons							
Dropout = 0.5								
Fully connected 2	4096-neurons							
Dropout = 0.5								
Fully connected 3	2622 classes							
Softmax	2622 classes							

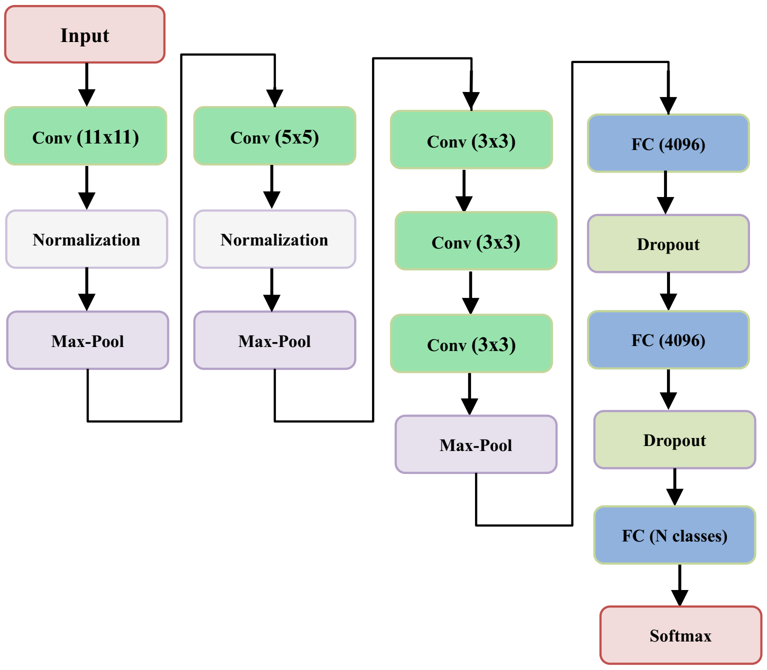


Fig. 6.2 Modified VGG-face architecture for face recognition

The output from the first pooling layer is convolved with a filter of size 5 $\times$ 5 with a stride of one and padding of two. A total of 256 filters are used to produce 256 feature maps of size 27 $\times$ 27. The feature maps are normalized and the results are subjected to max-pooling with stride of 2 which produces 256 feature maps of size 13 $\times$ 13. The feature maps are subjected to three convolutions with filter size of 3 $\times$ 3 and padding of 1. This is followed by third max-pooling with a stride of 2 which reduces the size of the 256 feature maps to 6 $\times$ 6.

Fully connected layers connect every neuron in one layer to every neuron in another layer. The result from the third pooling layer are passed to the first fully connected layer of size 4096, and then to the second fully connected layer of size 4096. Dropout technique is used after the first and second fully connected layers. It consists of setting the output of a hidden neuron to zero with a probability of 0.5. The third fully connected layer has size N (number of classes).

The results from the third fully connected layer are passed to Softmax layer for classification. The total number of class labels for the ORL, Faces94, extended-Yale, Yale, CVL, and FERET datasets are 40, 152, 38, 15, 114, and 20, respectively. With the outputs from the last fully connected layer represented as $z_1, z_2, \ldots, z_k$, the probability scores at Softmax layer are calculated using Eq. 6.1 as follows:

Table 6.2 Details of modified VGG-face architecture for face recognition

Layer name	Input size	Filter size	# Filters	Window size for pooling	Stride	Padding	Output size	# Feature maps
Conv1	224 × 224	11 × 11	64	–	4	0	54 × 54	64
Pooling1	54 × 54	–	–	3 × 3	2	0	27 × 27	64
Conv2	27 × 27	5 × 5	256	–	1	2	27 × 27	256
Pooling2	27 × 27	–	–	3 × 3	2	0	13 × 13	256
Conv3	13 × 13	3 × 3	256	–	1	1	13 × 13	256
Conv4	13 × 13	3 × 3	256	–	1	1	13 × 13	256
Conv5	13 × 13	3 × 3	256	–	1	1	13 × 13	256
Pooling3	13 × 13	–	–	3 × 3	2	0	6 × 6	256
Fully connected 1	4096-neurons							
Dropout = 0.5								
Fully connected 2	4096-neurons							
Dropout = 0.5								
Fully connected 3	N classes							
Softmax	(where $N = 40$ for ORL, 152 for faces-94 and 38 for extended-Yale, 15 for Yale, 114 for CVL, and 20 for FERET face datasets)							

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} \quad \text{for } j = 1, 2, \ldots, N \tag{6.1}$$

where $\sigma(z_j)$ represents scores.

For the ORL face dataset, there are a total of 40 classes. The probability distribution output of the 40 clasess is calculated as follows:

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{40} e^{z_k}} \quad \text{for } j = 1, 2, \ldots, 40.$$

The Softmax probabilities determine the target class for the given input. The target class is the one corresponding to which the highest probability is obtained. The main advantage of using Softmax is that output probabilities range from 0 to 1, and the sum of all the probabilities is equal to one.

6.3 Performance Comparison of Deep Learning Models for Face Recognition

The performance comparison of the two deep learning models, obtained as a result of training the two architectures described above, is provided in this section using face image databases. There are a number of face image databases that are available publicly. The databases that are used here for performance comparison are described below.

Olivetti Research Ltd., Face Database (ORL)
The Olivetti Research Ltd., Face Database (ORL) contains 400 face images of 40 different persons, with 10 different images of each person. These face images have been taken mainly with different facial expression (open eyes, closed eyes, smiling and not smiling faces). The face images are in PGM format of size 92 $\times$ 112 pixels with 256 gray levels. Figure 6.3 shows some sample images of one person from the ORL face database.

Yale Face Database
The Yale face database contains 165 ($11 \times 15 = 165$) images of 15 persons. Each person in the database has 11 different images taken mainly under varying lighting conditions (dark lighting conditions, bright lighting conditions, medium lighting conditions, varying lighting conditions using spectacles). Figure 6.4 shows some sample images of one person from the Yale face database.

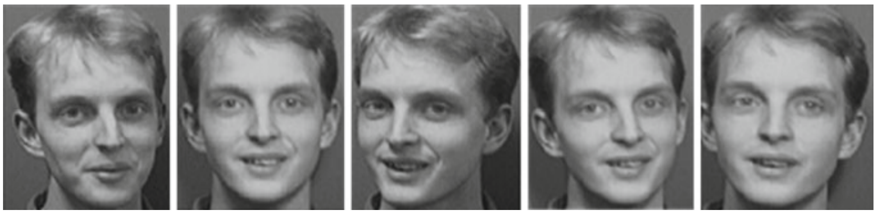


Fig. 6.3 Five images of a person in ORL face database showing different facial expressions

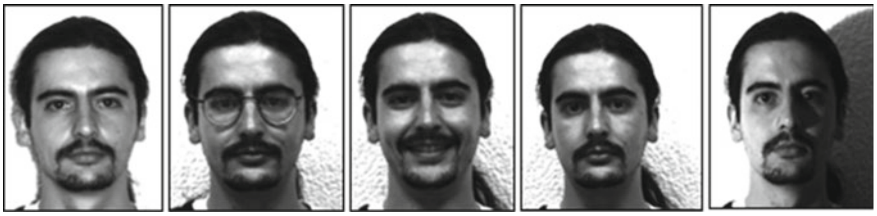


Fig. 6.4 Five face images of one person from the Yale face database showing variation in lighting condition

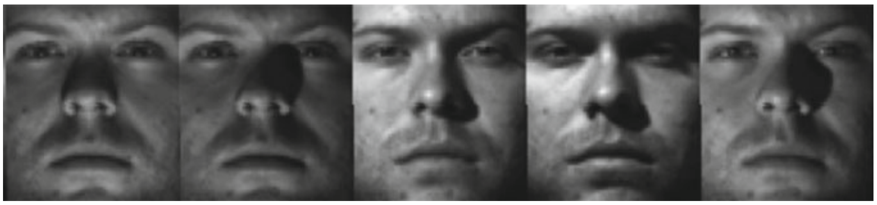


Fig. 6.5 Five face images of one person from the Yale-B face database showing variation in lighting condition

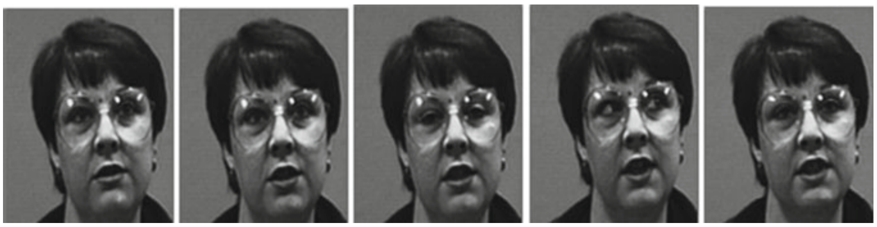


Fig. 6.6 Five face images of one person from the Faces94 face database showing variation in expression

Extended Yale-B Cropped Face Database

The extended Yale-B cropped face database contains 2470 ($65 \times 38 = 2470$) images of 38 persons. Each person in the database has 11 different images taken mainly under varying lighting conditions (dark lighting conditions, bright lighting conditions, medium lighting conditions). Figure 6.5 shows some sample images of one person from the cropped Yale-B face database.

Faces94 Database

The Faces94 database contains images of 152 persons. The resolution of each image is (180×200 pixels), and this face database contains three directories: female (20), male (112), and male staff (20). Figure 6.6 shows some sample images of one person from the Faces94 face database.

FERET Face Database

The FERET database contains a total of 11,338 face images. The images of this face database were collected by taking images of 994 persons at various angles. The images are of size 512×768 pixels and the files are in PPM format. Figure 6.7 shows some images of a person in the FERET database that were taken at various angles.

CVL Face Database

The CVL face database contains face images with varying poses. The total number of images in this database is 798 which belong to 114 persons. The size of the images is 640×480 and the format of the images is JPG. These images belong to 108 male persons and 6 female persons. Figure 6.8 shows 5 sample images from CVL face database.

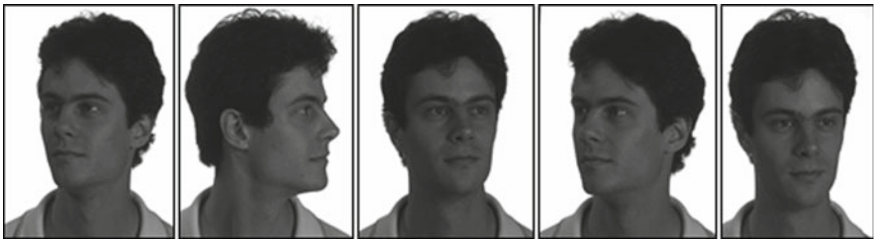


Fig. 6.7 Five face images of one person in FERET face database showing variation in poses

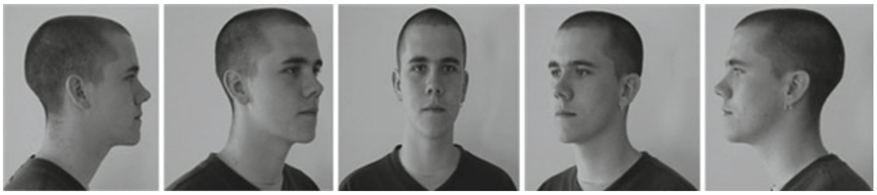


Fig. 6.8 Shows the face images of one person from the CVL face database showing variation in poses

The performance comparison of deep learning models and other traditional methods is carried out. The traditional methods used for comparison purposes are: Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Discrete Cosine Transform (DCT), Independent Component Analysis (ICA), Gabor Wavelet Transform (GWT), and Elastic Bunch Graph Matching (EBGM). The performance is evaluated by means of recognition accuracy results. In PCA, a vector that has the largest variance associated with the dataset is computed. On the other hand, LDA computes a vector in the underlying space that best discriminates among classes. However, if the training images in LDA are less; then, the scatter matrix is very difficult to evaluate. ICA shows good results than PCA when cosines were used as the similarity measure, however, the performance remains almost the same when Euclidean distance was used as a similarity measure. Although ICA shows better results than PCA in some databases, it is computationally more expensive than PCA. In DCT, energy of the original data may be concentrated in only a few low-frequency components of DCT depending on the correlation in the data. Also, low-frequency components normally contain the most information of the image; hence, if the value of the AC coefficients is higher, then it implies finer image information. Gabor wavelets are useful for image analysis because of their biological relevance and computational properties. In EBGM, features are represented by Gabor jets, where jets are extracted with manually selected landmark locations; and the model jets are then collected in a data structure which is called a bunch graph.

The performance of the deep learning models and the traditional methods, i.e., PCA, LDA, DCT, ICA, GWT, and EBGM algorithms for the face recognition task is compared by using the publicly available face image databases discussed above.

6.3.1 Performance Comparison with Variation in Facial Expression

The performance comparison of the deep learning models and six major traditional methods for face recognition on images with variation in facial expressions was done using the ORL and the Faces94 face databases. The ORL face database was divided into two sets: a training set and a testing set. The training set contained 200 images of 40 individuals with 5 different images of each person (i.e., $40 \times 5 = 200$) and the test set also contained 200 images of 40 individuals images with 5 different images of each person (i.e., $40 \times 5 = 200$). Similarly, the Faces94 dataset was divided into a training set and a testing set. The training set contained 760 face images of 152 different subjects and the testing set contained different 760 face images of these 152 subjects. The results obtained after performing the tests on ORL and Faces94 face databases are given in Tables 6.3 and 6.4. The VGG-face model and the modified VGG-face model results are better for images varying in expression than the results obtained by the traditional methods. Figure 6.9 shows the graphical representation of the performance comparison of results of the deep learning models and six major traditional methods for face images varying in expression.

Table 6.3 Performance comparison of the deep learning models and six major traditional methods on ORL database having face images with variation in facial expression

Strategy	Results	
	Method	Recognition rate in %
Using first five images of each person for training and remaining five images of that person for testing	PCA	85.50
	LDA	88.50
	DCT	91.50
	ICA	87.50
	GWT	69.50
	EBGM	91.50
	VGG-face model	99.00
	Modified VGG-face model	98.00

Table 6.4 Performance comparison of the deep learning models and six major traditional methods on Faces94 database having face images with variation in facial expression

Strategy	Results	
	Method	Recognition rate in %
Using first five images of each person for training and remaining five images of that person for testing	PCA	96.40
	LDA	97.00
	DCT	98.50
	ICA	80.00
	GWT	46.80
	EBGM	91.00
	VGG-face model	98.80
	Modified VGG-face model	99.21

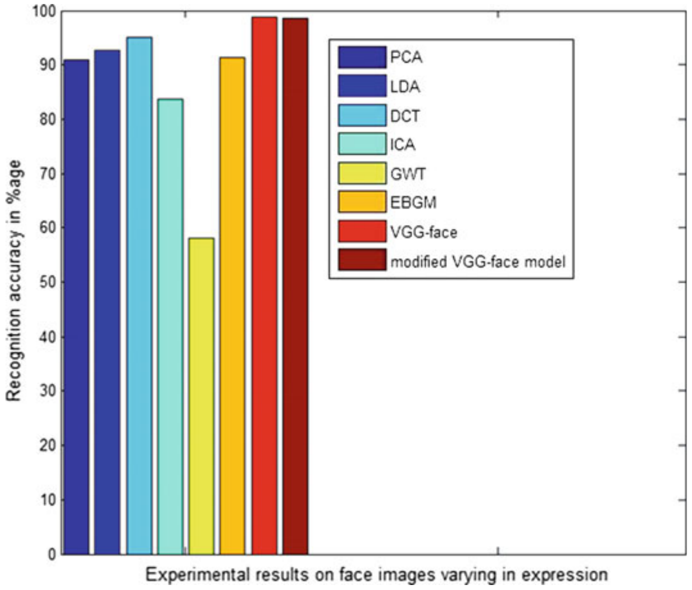


Fig. 6.9 Shows the performance of deep learning models and six major traditional methods for face recognition on facial images varying in expression

6.3.2 Performance Comparison on Images with Variation in Illumination Conditions

The performance comparison of the deep learning models and six major traditional methods for face recognition on images with variation in illumination conditions was done using Yale and extended Yale-B face databases. The Yale database was

Table 6.5 Performance comparison of deep learning models and six major traditional methods for face recognition on Yale face database with images varying in illumination

Strategy	Results	
	Method	Recognition rate %
Using first five images of each person for training and remaining six images of that person for testing	PCA	82.22
	LDA	86.66
	DCT	76.66
	ICA	71.11
	GWT	88.80
	EBGM	91.11
	VGG-face model	98.67
	Modified VGG-face model	97.33

Table 6.6 Performance comparison of the deep learning models and six major traditional methods for face recognition on extended Yale-B face database with images varying in illumination

Strategy	Results	
	Method	Recognition rate %
Using first five images of each person for training and remaining five images of that person for testing	PCA	86.84
	LDA	87.89
	DCT	70.00
	ICA	70.00
	GWT	78.07
	EBGM	65.78
	VGG-face model	85.30
	Modified VGG-face model	89.47

divided into two sets: training set and the testing set. The training set had 75 images (5×15) of 15 persons with 5 different images of each individual, and the testing set had 90 images ($6 \times 15 = 90$) of same 15 persons with 6 different images of each individual. The Yale-B dataset was divided into 2 sets where training set contained 190 face images of 38 different persons with 5 face images of each person and the testing set contained different 190 face images of the same 38 persons with 5 face images of each person. The results of recognizing face images of Yale database and Yale-B database using deep learning models and six major traditional methods are shown in Tables 6.5 and 6.6. The VGG-face model and modified VGG-face model produced similar results on Yale database, however, for the Yale-B face database, the modified VGG-face model shows better results as compared to VGG-face and the six traditional methods. Figure 6.10 shows the graphical representation of the performance comparison of recognizing Yale and Yale-B face images with varying

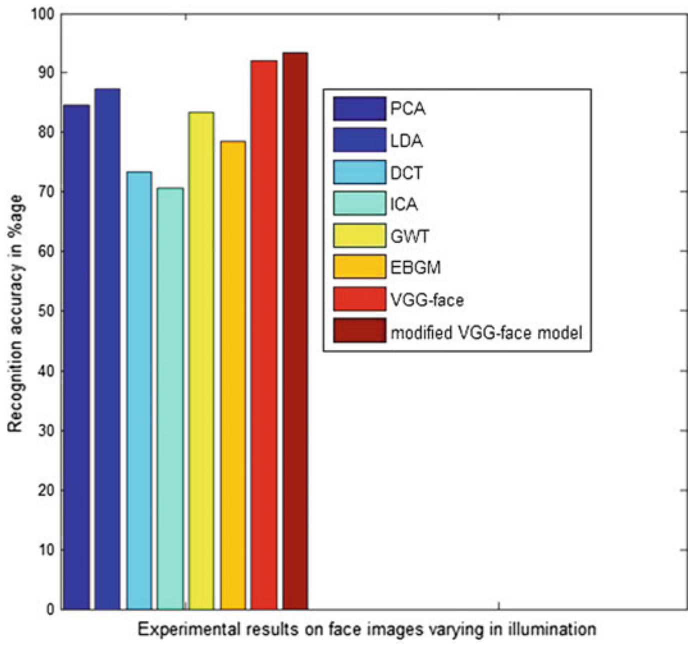


Fig. 6.10 Shows the performance comparison of deep learning models and six major traditional methods for face recognition on facial images varying in illumination

illumination by the modified VGG-face model, VGG-face model and by the six traditional methods.

6.3.3 Performance Comparison with Variation in Poses

The performance comparison of the deep learning models and six major traditional methods for face recognition on images with variation in poses was done using the FERET and CVL databases. Two-hundred images of 20 individuals from the FERET face images with varying poses were divided into 2 sets: a training set and a test set. One-hundred images ($20 \times 5 = 100$) with 5 different images of each individual were used for training and 100 images with 5 different images of each individual were used for testing ($20 \times 5 = 100$). Similarly, the CVL database was divided into two sets: training set and testing set. The training set contained 456 face images of 114 individuals with 4 face images of each person and the testing set contained 342 face images of the same 114 individuals with 3 different images of each person. The results of recognizing face images of the FERET database with deep learning models and six major traditional methods are given in Tables 6.7 and 6.8. The modified VGG-face model has shown better results than the VGG-face model and the six traditional

Table 6.7 Performance comparison of deep learning models and six major traditional methods for face recognition on FERET face database with images varying in poses

Strategy	Results	
	Method	Recognition rate %
Using five images of each person for training and five images for testing	PCA	68.00
	LDA	60.00
	DCT	67.00
	ICA	60.00
	GWT	44.00
	EBGM	77.00
	VGG-face	76.00
	Modified VGG-face model	90.00

Table 6.8 Performance comparison of the deep learning models and six major traditional methods for face recognition on CVL face database with images varying in poses

Strategy	Results	
	Method	Recognition rate %
Using four images of each person for training and three images for testing	PCA	64.00
	LDA	65.60
	DCT	65.00
	ICA	46.66
	GWT	40.94
	EBGM	64.28
	VGG-face	60.52
	Modified VGG-face model	94.44

Table 6.9 Performance comparison of the modified VGG-face model and the VGG-face model for recognition of face images varying in expression, illumination and poses

Strategy	Results	
	Method	Recognition rate in %
Using five images of each person for training and five images for testing	VGG-face	79.20
	Modified VGG-face model	96.67

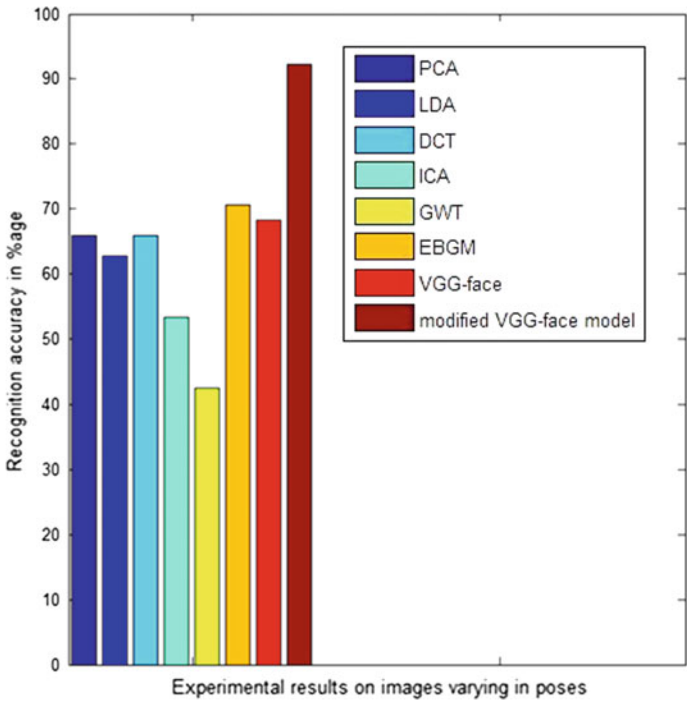


Fig. 6.11 Shows the performance comparison of deep learning models and six major traditional methods for face recognition on facial images varying in poses

methods for face recognition. Figure 6.11 shows the graphical representation of the performance comparison of the deep learning models and the six traditional methods on images of varying poses.

Finally, the overall comparison of the deep learning models (VGG-face model and the modified VGG-face model) on the combined datasets (all six datasets, i.e., ORL, Faces94, Yale, Yale-B, FERET, and CVL) is given in Table 6.9. The 6 datasets were divided into 379 classes. There were 1895 images in the training set and different 1800 images of the same subjects in the testing data. The results obtained after performing the experiments on the whole dataset of 379 classes containing face images varying in expression, illumination, and poses are given in Table 6.9. The modified VGG-face model for face recognition showed better results than other methods for recognition of face images varying in expression, illumination, and poses. Figure 6.12 shows the graphical representation of the comparison (Fig. 6.13).

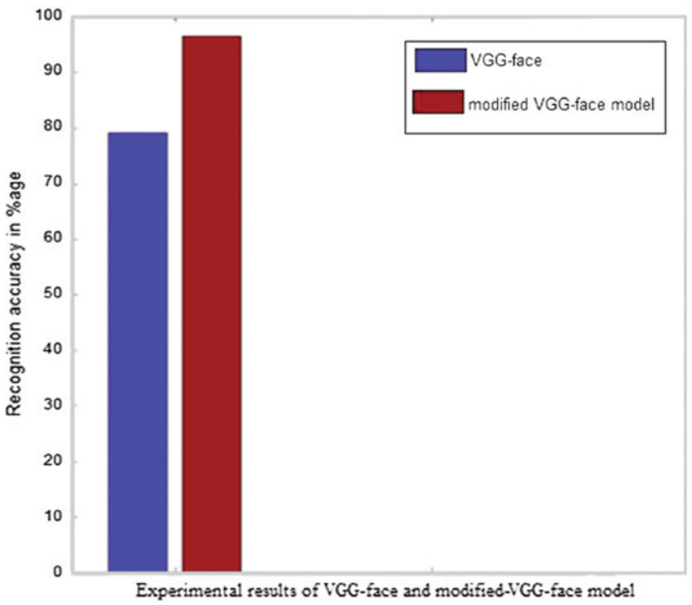


Fig. 6.12 Shows the performance comparison of the modified VGG-face model and the VGG-face model for recognizing facial images varying in expression, illumination, and poses

6.4 Challenges and Future Research Direction

A number of deep learning architectures have been used for face recognition. A deep learning model can run faster if the size of the filters and the network is reduced. A comparison of the results of the two models, i.e., VGG-face and modified VGG-face models used here indicate that the use of small-sized filters do not necessarily produce good results for the face recognition task. The challenge lies in defining the optimal deep learning architecture and hyper-parameters that is fast and accurate for face recognition task.

The performance of a deep learning model can be improved if image enhancement is performed before the convolution operation to make the appearance of blurred or low-resolution images better. Missing data is a challenge for deep learning networks and the performance of the network deteriorates appreciably when large contiguous areas of images are missing. This observation recommends that research into deep convolutional neural networks capable of recognizing objects from partially observed data is required and can be a focus of future research efforts.

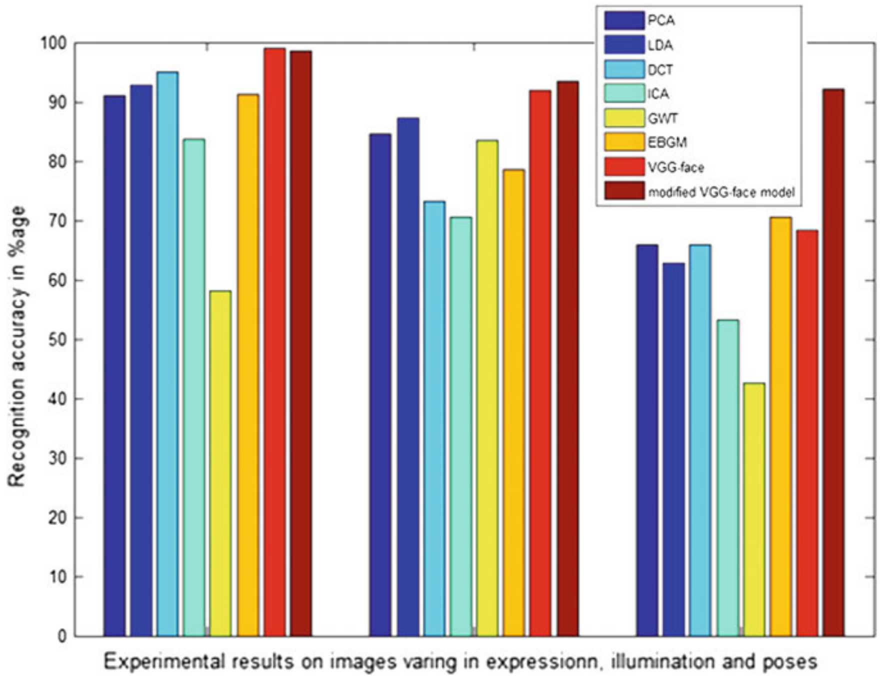


Fig. 6.13 Shows performance comparison of the modified VGG-face model, the VGG-face, and the six traditional methods. The modified VGG-face model performs better than all other models

Bibliography

Bhat, F.A., Wani, M.A.: Performance comparison of major classical face recognition techniques. In: 2014 13th International Conference on Machine Learning and Applications (ICMLA), IEEE (2014)

Bhat, F.A., Wani, M.A.: Face recognition using convolutional neural network. In: 2017 4th International Conference on Computing for Sustainable Global Development (INDIACom), pp. 460–464, IEEE (2017)

Bhat, F.A., Wani, M.A.: A robust face recognition model based on convolutional neural networks. J. Artif. Intell. Res. Adv. **5**(1), 1–11 (2018)

Parkhi, O.M., Vedaldi, A., Zisserman, A.: Deep face recognition. BMVC **1**(3) (2015)

Sun, Y., Wang, X., Tang, X.: Deep learning face representation from predicting 10,000 classes. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2014a)

Sun, Y., et al.: Deep learning face representation by joint identification-verification. In: Advances in Neural Information Processing Systems (2014b)

Sun, Y., Wang, X., Tang, X.: Deeply learned face representations are sparse, selective, and robust. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2015)

Chapter 7
Supervised Deep Learning in Fingerprint Recognition

7.1 Introduction

Fingerprint recognition refers to the process of identifying or confirming the identity of an individual by comparing two fingerprints. Fingerprint recognition is one of the most researched and reliable biometric techniques for identification and authentication. Any system which uses image processing techniques to automatically perform the process of obtaining, storing, analyzing, and matching of a fingerprint with another fingerprint and generating the match is called **Automatic Fingerprint Identification System (AFIS)**. It is a system which takes a fingerprint and picks the most likely matches from millions of fingerprint images stored in the database. With the growth in technology, many algorithms and methods have been proposed so far to automatically match the fingerprints without any human interference or assistance.

7.2 Fingerprint Features

A fingerprint consists of a number of ridges and valleys. Ridges are the upper skin layer segments (crest) of the finger and valleys are the lower segments and these ridges run in parallel (Fig. 7.1), but there exist one or more regions where they assume distinctive shapes and these regions are called *singularities* or *singular regions/points*. Fingerprint features are divided into three levels: level 1 (patterns like singular points), level 2 (minutiae points like bifurcation and ridge endings), and level 3 (pores and contours) features.

Level 1 (pattern) features are the global features which include the general ridge flow and patterns like core and delta location. Singular points are the most important global characteristics of a fingerprint and there are mainly five types of singularities in fingerprints as shown in Fig. 7.2.:

- Arch: Ridges form an arc shape which rises at center (Fig. 7.2a).
- Left Loop: Ridges form a curve at center forming a loop toward left (Fig. 7.2b).

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_7

Fig. 7.1 Example of a fingerprint image

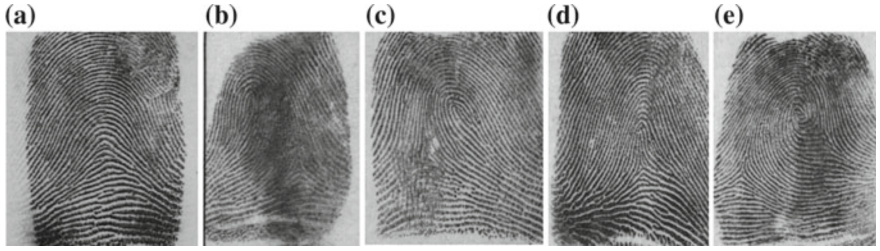


Fig. 7.2 Fingerprint classes **a** arch, **b** left loop, **c** right loop, **d** tented arch, **e** whorl

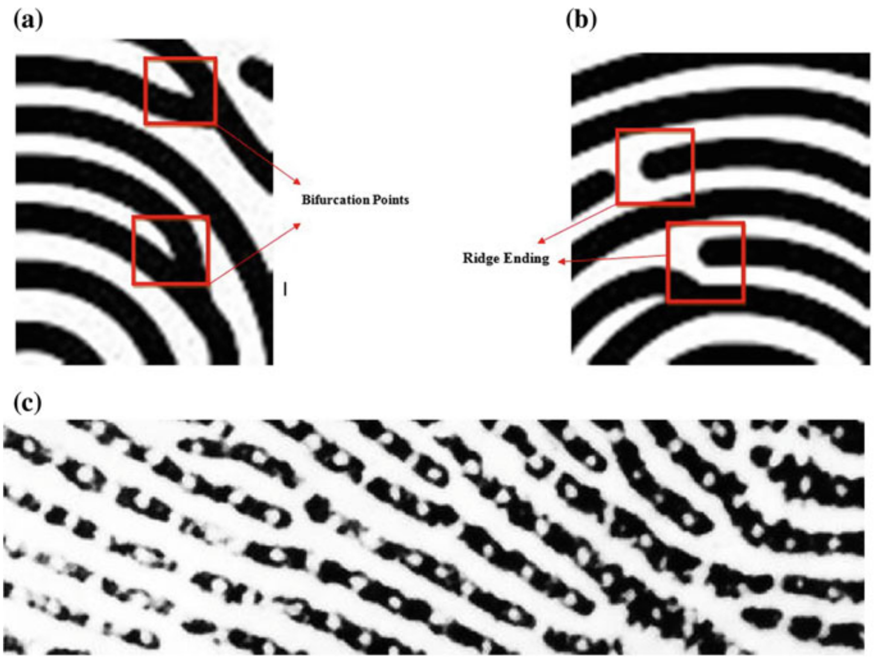


Fig. 7.3 **a** bifurcation point (level 2 feature), **b** ridge ending (level 2 feature) and **c** pores, ridge contours (level 3 features) of a fingerprint

- Right Loop: Ridges form a curve at center forming a loop toward the right (Fig. 7.2c).
- Tented Arch: Ridges form an arc, have an angle, an upthrust shape which rises at center (Fig. 7.2d).
- Whorl: Ridges form circularly around a central point (Fig. 7.2e).

Level 2 (minutiae points): In addition to different patterns, fingerprint ridges contain various minute information called minutia points which are very useful in matching fingerprints and are most commonly used in automatic fingerprint matching. Minutia actually means minute details and in the context of fingerprints, it means numerous ways a ridge can be disjointed and irregular. There are around 150 types of minutia, out of which ridge ending and ridge bifurcation are most widely used because all other types are the combination of these two.

- Ridge Ending: A point where a ridge terminates or suddenly ends (Fig. 7.3a).
- Ridge bifurcation: A point where a ridge divides into two ridges (Fig. 7.3b).

Minutiae points are very important in fingerprints recognition because these points remain unchanged during person's lifetime.

Level 3 (pores and ridge contours) features include all dimensional properties of a ridge, such as sweat pores, ridge edges, width, incipient ridges, etc. Extraction of level 3 features from a fingerprint image requires high-resolution images (image resolution of 1000 ppi or more) as compared to the current standard of 500 ppi.

7.3 Automatic Fingerprint Identification System (AFIS)

Minutia-based fingerprint matching is the most popular and widely used approach for both human expert and automatic recognition systems. Minutia-based approaches have many advantages which make them suitable for automatic matching. These approaches require less space because fingerprints are not represented as an image but as a set of points (minutia points) in two-dimensional plane. Minutia points are extracted from two fingerprints and stored as a set of points which take very less space as compared to images. Also, matching two sets of points is much faster as compared to matching two images pixel by pixel. A minutia-based fingerprint recognition system usually consists of two main stages: (i) Feature Extraction Stage, (ii) Fingerprint Matching Stage. Feature extraction stage itself consists of many substages which include image segmentation, minutiae, and singular point extraction and classification, etc. Figure 7.4 below shows the block map of minutia-based fingerprint recognition system.

7.3.1 Feature Extraction Stage

In feature extraction stage, different useful features such as singular points and minutia points are extracted from the fingerprint image. Ridge bifurcation and ridge ending are the two important minutiae points and almost all the minutiae-based fingerprint recognition systems use at least these two minutia types. The accuracy of a fingerprint recognition system depends on the accuracy of feature extraction stage.

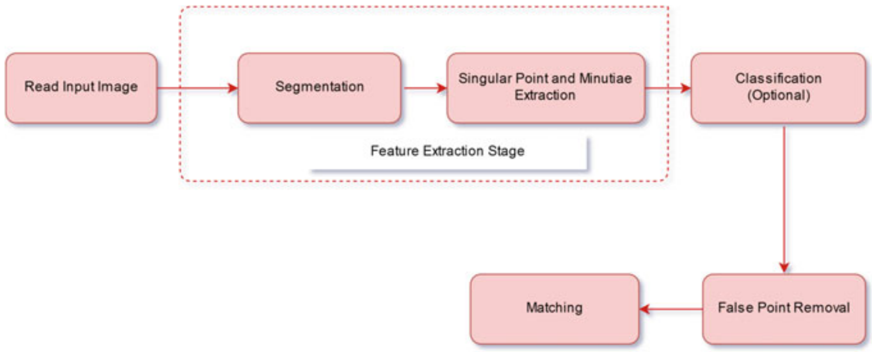


Fig. 7.4 Block map of automatic fingerprint recognition system

Feature extraction stage consists of many steps which include:

- ***Image Enhancement***:
  The first important thing in fingerprint recognition system is the quality of fingerprint image. The performance of an automatic fingerprint recognition system depends a lot upon the image quality and the preprocessing steps used. Fingerprint images extracted from various sources usually lack sufficient clarity and contrast. Hence, image enhancement is necessary and a major step in AFIS. Image enhancement is used to remove the noise from fingerprint image with the help of various techniques like Histogram, Gaussian smoothing filter, etc.
- ***Segmentation***:
  Fingerprint segmentation is the process of extracting Region of Interest (ROI) from a fingerprint image. It is carried out by removing the background from a fingerprint image.
- ***Minutiae Extraction***:
  After an obtaining enhanced segmented image, minutiae extraction is done using the following two main techniques:

  (1) *Grayscale-Based Extraction*:
  Grayscale-based extraction uses ridge tracing in which, starting at a point, ridges are traced by sailing along the ridges in a grayscale image and all the occurrences where a ridge ends or bifurcates are recorded.

  (2) *Binarization Based Extraction*:
  In binarization based extraction, the fingerprint image is first transformed into 1-bit image with 0-value for ridges and 1-value for furrows or valleys. Binary image consists of only two values 0 for black color and 1 for white color. The grayscale image is transformed to black and white by comparing each pixel value to a threshold value. If the pixel value is lower than the threshold value, the pixel value is assigned black otherwise it is assigned white. The binarized image is then thinned to obtain an image with all the ridges just 1 pixel wide followed by minutia points extraction by comparing each pixel along with its neighboring pixels of the image with predefined specified templates.

- ***False Minutiae Removal***:
  Any false minutia points introduced due to inadequate ink, ridge cross-connections, over inking and due to limitations of minutiae extraction techniques should be deleted for better reliability of the system. Some of these false minutia points are deleted by removing all those points that are too close to each other.

7.3.2 Minutia Matching Stage

After the extraction of features from fingerprints, matching is done to obtain the matching score. Fingerprint matching is a challenging task due to variations in the minutia details of different impressions of the same finger. The variations are due to displacement, rotation, nonlinear distortion, pressure, skin condition, etc., and as well as due to any errors that may be associated with feature extraction. There are various techniques available for fingerprint matching, some are based on using global features, some use local features, and some use both.

7.4 Deep Learning Architectures for Fingerprint Recognition

Deep learning, especially Convolutional Neural Network (CNN) has made tremendous success in the field of computer vision and pattern recognition as it does not require handcrafted feature extraction. Deep learning automatically learns features and structures under a sufficient number of training data. These advantages of CNNs make it suitable for various tasks in automatic fingerprint recognition/identification system: including segmentation, classification, feature extraction (minutiae points and singular points), ridge orientation estimation, etc. In next subsections, the application of deep learning in fingerprints recognition will be discussed.

7.4.1 Deep Learning for Fingerprint Segmentation

Fingerprint segmentation is the process of decomposing a fingerprint image into two regions: foreground region also known as Region of Interest (ROI) consisting of fingerprint area and background region, which consists of other irrelevant content like noise, etc. Fingerprint segmentation involves demarcation of all the foreground regions accurately in a fingerprint image while discarding all irrelevant contents. Accurate fingerprint segmentation is an important and critical step in automatic fingerprint recognition systems as it affects the reliable feature extraction and eventually, the overall performance of the system. Although significant progress has been made on automatic segmentation of plain/rolled fingerprints, latent fingerprint segmentation remains one of the most difficult tasks in automatic latent fingerprint recognition due to poor quality of images and complex background. Latent fingerprints can be present on any surface like glass, cup, newspaper, table, etc., and very often these surfaces are not clear or regular, thus making it difficult to extract fingerprint from these surfaces. Their ridge structure is not clean and contains stains, spikes, lines, text, etc., thus making the segmenting of foreground regions very difficult. For poor quality images and noisy background, patch-based segmentation approach is the preferred over other techniques. Patch-based segmentation techniques are computationally expensive and slow but very useful in situations where the input image is distorted and background is noisy.

In patch-based segmentation method, segmentation problem is posed as a classification problem. The input image is divided into fixed size patches, and then these patches are fed to a classifier and only positive patches are assembled to form the segmented image.

Segmentation is done by dividing the fingerprint image into blocks, followed by block classification based on gradient and variance information. The patch-based segmentation technique for fingerprint images using Convolutional Neural Networks consists of the following four modules:

- Splitter (*S*)
- Classifier (*C*)
- False Patch Normalizer (*F*)
- Patch Assembler (*A*)

The Splitter (*S*) module divides an input image into equal size blocks called patches. The Classifier (*C*) module uses CNN model which classifies each of these patches into fingerprint and non-fingerprint patches. The False Patch Normalizer (*F*) module corrects the misclassified and isolated patches. The Patch Assembler (*A*) module reassembles these patches and generates a segmented image. The block diagram of the patch-based segmentation method is shown in Fig. 7.5. The convolution model used for patch classification is trained on both plain and latent fingerprints, thus the technique is suitable for both plain and latent fingerprints.

7.4.1.1 Convolutional Neural Network (CNN) as Patch Classifier

(i) **CNN Patch Classifier Architecture**
Convolutional Neural Network (CNN or ConvNet) based patch classifier is used to classify each patch into fingerprint and non-fingerprint patch. The patch classifier consists of three convolutional layers, one subsampling layer, two fully connected layers, and one dropout layer as shown in Fig. 7.6. The output of the last fully connected layer is fed to a 2-way Softmax classifier, which classifies a patch into one of the two types. The first convolutional layer convolves the 16 $\times$ 16 input image block with 64 filters of size 5 $\times$ 5 with a stride of 1 pixel and padding 2 producing 64 feature maps of size 16 $\times$ 16. Each convolution layer is followed by a Rectified Linear Units (ReLU) layer. Max-pooling is performed after the first convolution operation which produces 16 feature maps of size 8 $\times$ 8. The pooled output of 16 feature maps of size 8 $\times$ 8 is fed to the second convolutional layer which has 64 filters of size 5 $\times$ 5. The second convolution layer produces 16 feature maps of size 8 $\times$ 8. The 16 feature maps of size 8 $\times$ 8 are fed to the third convolutional layer which convolves it with 256 filters of size 5 $\times$ 5. The output is passed to a fully connected layer with 256 neurons followed by dropout layer. Finally, the network has fully connected layer with Softmax classifier producing binary classification. Figures 7.7 and 7.8 shows the operation of the first convolution layer and fully connected layer, respectively.

(ii) **CNN Patch Classifier Training**
The CNN patch classifier was trained on around 10,000 fingerprint and non-fingerprint patches prepared from IIIT-D latent fingerprint database. Since the

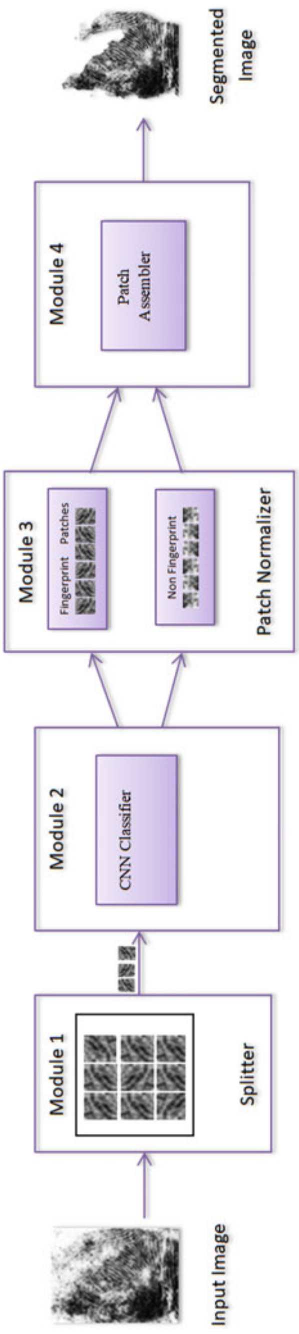


Fig. 7.5 Block diagram of patch-based fingerprint segmentation technique

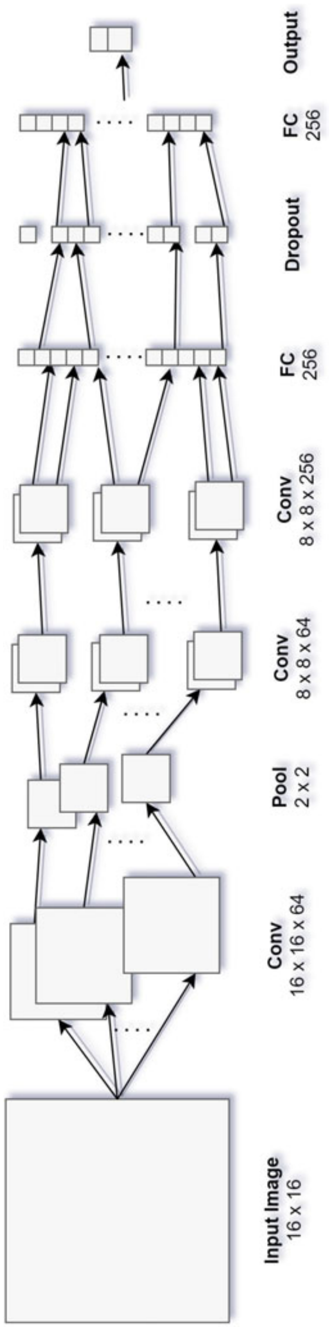


Fig. 7.6 Architecture of CNN patch classifier

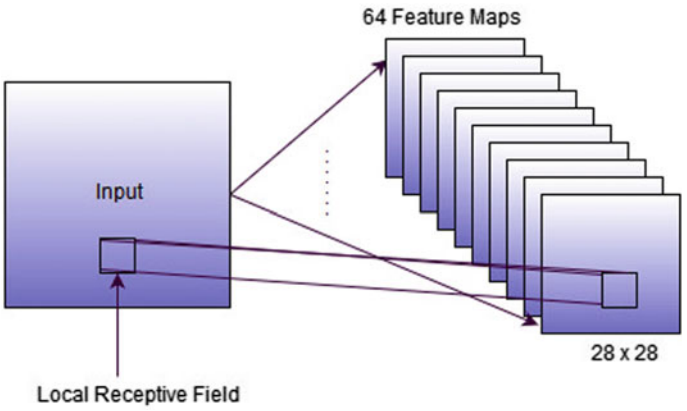
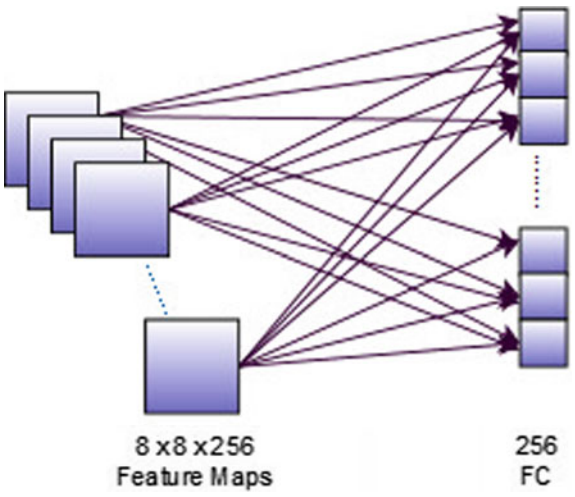


Fig. 7.7 First convolutional layer

Fig. 7.8 Fully connected Layer



training dataset is not sufficient to train CNN model from scratch, the weights were initialized using VGG-16 model. The CNN model was then fine-tuned with 10,000 image patches using Stochastic Gradient Descent (SGD) with learning rate of 0.0001, batch size of 50. and weight decay of 1.

7.4.1.2 False Patch Normalizer

A low score associated with classifying a patch indicates that there is a possibility that the patch may have been misclassified. A misclassified patch is also referred to as a false patch. To reduce the number of false patches, a technique called "majority of neighbors" is used to decide the final label of the patch. Since all the patches of a

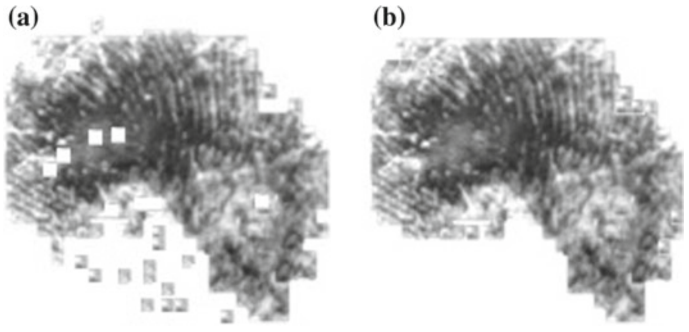


Fig. 7.9 Segmentation result **a** before and **b** after applying false patch removal technique

specific class are likely to be in the same neighborhood, the probability that a patch belongs to the class of majority of its neighbors is high. For each suspicious patch (i.e., whose score is less than a defined threshold), the class label of its four neighboring patches is checked. If at least three neighboring patches are of the same class as the patch under test, then it is accepted as true patch label, otherwise, it is treated as a false patch and its class label is changed from fingerprint to non-fingerprint or vice versa depending upon the actual class.

For example, if p_i is ith patch and n_1, n_2, n_3, and n_4 are its four neighbors, then

$$\text{label}(p_i) = \text{majority}(\text{label}(n_1), \text{label}(n_2), \text{label}(n_3), \text{label}(n_4))$$

The method "*Majority of neighbors*" performed well on low-quality latent fingerprint images and reduced the false patches by around 20%. The effect of the method is shown in Fig. 7.9, which shows segmentation result before and after removing false patches.

7.4.1.3 Patch Assembler

The Patch Assembler takes the classified patches as input, discards negative patches and assembles the positive patches (classified as fingerprint patches) to produce the segmented image. Figure 7.10 shows the final result of the segmentation technique on latent fingerprints. The left column contains input images and right column their corresponding segmented image.

7.4.1.4 Performance Evaluation

The patch-based segmentation technique results are discussed below on IIIT-D latent fingerprint images. To measure the accuracy and precision, the following metrics are used:

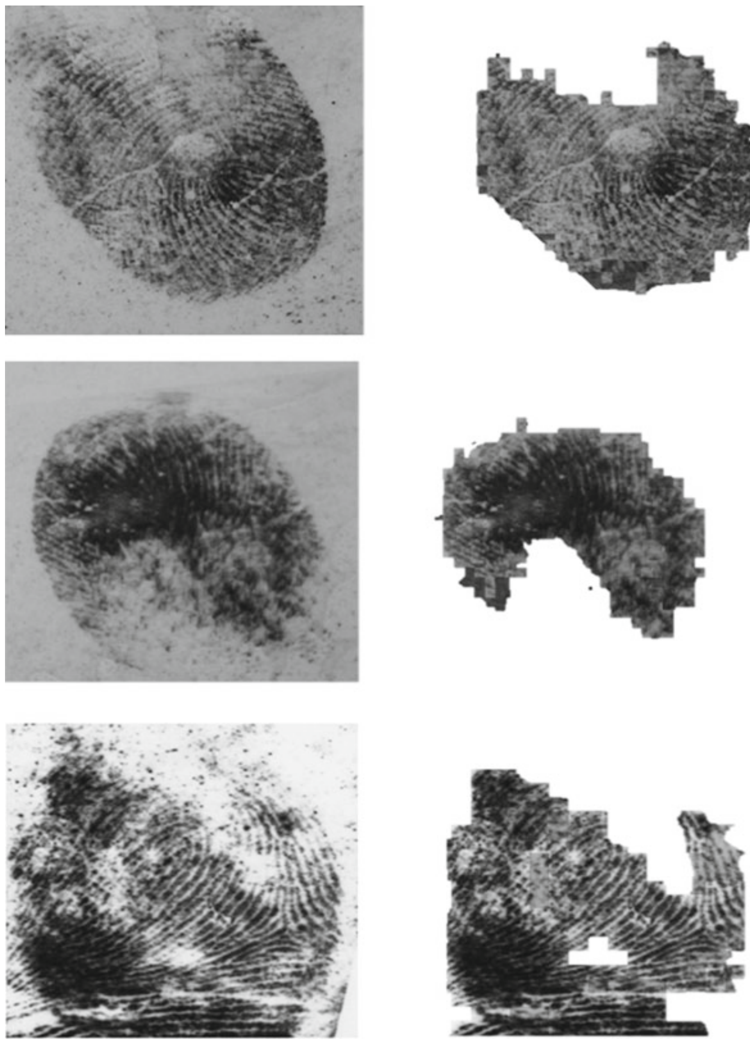


Fig. 7.10 Result of patch-based segmentation technique on latent fingerprints from IIIT-D latent database. The left column contains input images and right column their corresponding segmented image

- **True Positive (TP)**: Number of patches correctly classified as belonging to true class.
- **True Negative (TN)**: Number of patches correctly classified as not belonging to true class. It is equivalent to correct rejection.
- **False positive (FP)**: It is type-1 error and equivalent to false alarm.
- **False Negative (FN)**: It is type-1 error and equivalent to miss.

Table 7.1 Confusion matrix of patch classifier

Total number of images 2346	True positives (TP) 1022	True negatives(TN) 1065
	False negatives(FN) 196	False positives(FP) 63

Table 7.1 gives the above metrics in the form of confusion matrix.

The above metrics are used for calculating accuracy, precision, and recall are shown below:

Accuracy: It is simply the ratio of correctly predicted observation of the total observations. The mathematical formula is given as

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}} = \frac{2087}{2346} = 88.95$$

Therefore, **Accuracy = 88.95%**

Precision: It is the fraction of relevant instances among the retrieved instances. The formula is given as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{1022}{1085} = 94.19$$

Precision = 94.19%

Recall: It is the fraction of relevant instances that have been retrieved over total relevant instances in the image. It is based on an understanding and measure of relevance.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{1022}{1218} = 83.90$$

Recall = 83.90%

Performance comparison of different segmentation techniques is given in Table 7.2. The table shows False Detection rate (FDR) and Missed Detection Rate (MDR) of different segmentation techniques. The CNN-based technique has been tested on IIIT-D latent database only, with FDR of 5.2% and MDR of 13.8%. The second best method tested on the same database has achieved FDR and MDR of 18.7% and 9.22%, respectively (Fig. 7.11).

7.4.2 Deep Learning for Fingerprint Classification

Fingerprint classification plays an important role in the Automatic Fingerprint Identification System (AFIS) as it effectively reduces the database size as well as match-

Table 7.2 Performance comparison of segmentation methods

Approach	Database	FDR %	MDR %	Avg.
Ridge orientation and frequency computation	NIST SD27	47.99	14.78	31.38
Adaptive total variation	NIST SD27	26.13	14.10	20.12
K-means clustering	NIST SD27	26.06	4.77	15.42
Fractal Dim and WELM	NIST SD27 IIIT-D (Good Quality)	18.7 10.07	9.22 6.38	13.96 8.23
CNN method	IIIT-D IIIT-D (Good Quality)	5.2 4.7	13.8 10.5	9.2 7.6

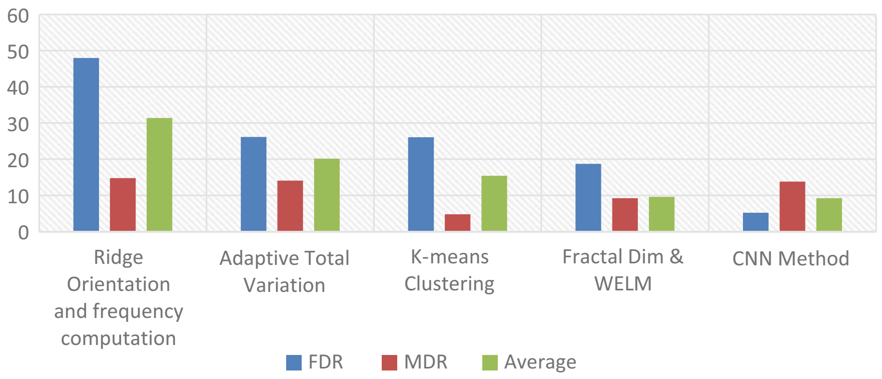


Fig. 7.11 Performance Comparison of various Latent Fingerprint segmentation techniques

ing time. Fingerprints can be broadly divided into five different classes: (a) Arch, (b) Left Loop, (c) Right Loop, (d) Whorl, and (e) Tented Arch. Figure 7.2 shows the images of five different fingerprint classes from the National Institute of Standards and Technology (NIST) database. Fingerprint classification is one of the important steps in automatic fingerprint recognition systems as it can substantially bring down the number of comparisons at the time of matching.

The CNN-based technique, which is called CNN-AFC classifies a fingerprint into five different classes, and then extracts the singular point from the fingerprint image. It uses a 22-layer CNN model trained on NIST-DB4 dataset for classification of fingerprints.

7.4.2.1 CNN Architecture for Fingerprint Classification

The CNN architecture (CNN-AFC) for classification of fingerprints consists of 22 layers including 5 convolutional layers, three pooling layers, three fully connected layers, and one dropout layer. The output of the last fully connected layer is fed to a 5-way softmax classifier which classifies the input into one of the five labels. The block diagram and architecture of the CNN model are shown in Figs. 7.12a, b respectively.

The first convolutional layer convolves the 224 × 224 × 3 input image block with 64 filters of size 11 × 11 × 3 with a stride of 4 pixels and 0 padding producing 64 feature maps. A Rectified Linear Units (ReLU) layer follows each convolution layer. Max-pooling layer follows the first, second, and fifth convolutional layers. The second convolutional layer takes as input the normalized and pooled output of the first convolutional layer and convolves it with 256 filters of size 5 × 5. The third convolutional layer takes input from the normalized output of the second convolutional layer and convolves it with 256 filters of size 3 × 3. Third and fourth convolutional layers are connected without any pooling or normalization layer and both these layers have 256 filters of size 3 × 3. In addition to the abovementioned layers, the ConvNet also contains dropout and batch normalization layers to overcome the problem of overfitting. ***Dropout*** refers to dropping out units in a neural network. Dropping a unit means temporarily disconnecting it from the network including all its incoming and outgoing connections. The dropped out units neither contribute to the forward pass nor do they contribute in backpropagation. By using dropout, the network is forced to learn more robust features as network architecture changes with each input.

Batch Normalization
Normalization is the process of scaling down the data to some reasonable limit to standardize the range of features. Batch Normalization (BN) is applied after the first two convolutional layers to remove covariate shift and reduce training time. Also, it has been observed that BN reduces the effects of exploding and vanishing gradients because everything becomes roughly normally distributed.

Batch Normalization is mathematically given as

$$x_i^* = x_i * \frac{M(x)}{\sqrt{V(x)}}$$

where x_i^* is the new normalized value of x_i, $M(x)$ is mean within a batch and $V(x)$ is its variance within a batch. In Convolutional Neural Network, every layer/filter is normalized, i.e., every generated value is treated as a value for normalizing. If the batch size is N and the output (feature map) generated by the convolution has a width of W and height of H, then the mean (M) is calculated over $N * W * H$ values (same for the variance). Batch normalization layer follows the first and second convolutional layers. The dropout layer is used after the first fully connected layer of the CNN model.

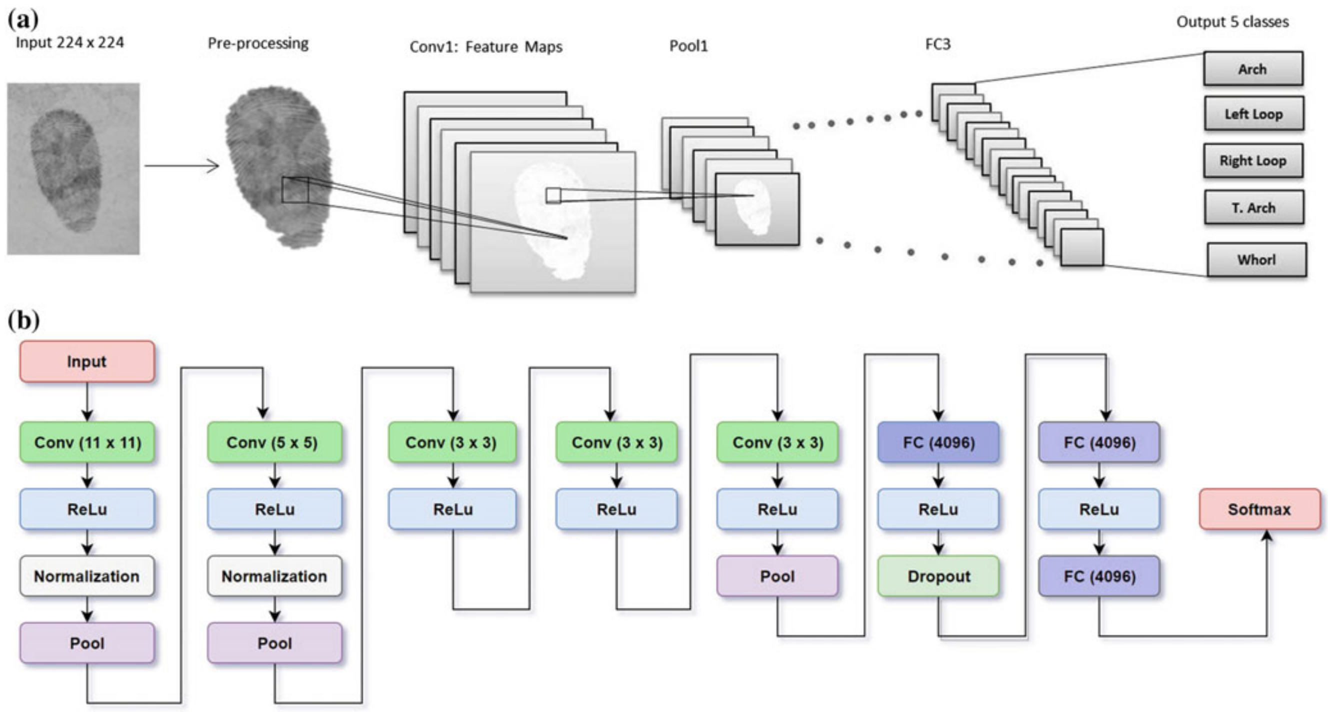


Fig. 7.12 **a** Block diagram of ConvNet model in CNN-AFC. **b** Architecture of ConvNet model in CNN-AFC

Table 7.3 Classification result of CNN-AFC Model on IIIT-D Latent fingerprint database

Actual class	Predicted class					Accuracy 78.2% (%)
	A	*L*	*R*	*T*	*W*	
A	97	5	2	22	0	76.98
L	13	107	3	5	9	78.1
R	11	1	103	5	13	77.4
T	24	3	5	90	0	73.7
W	0	11	8	0	109	85.1

Table 7.4 Classification result of CNN-AFC Model on NIST-DB4 fingerprint database

Actual class	Predicted class					Accuracy 92.2% (%)
	A	*L*	*R*	*T*	*W*	
A	361	5	2	32	0	90.25
L	2	376	2	15	5	94.0
R	8	0	369	18	5	92.25
T	34	9	5	352	0	88.0
W	0	8	6	0	386	96.5

7.4.2.2 Performance Evaluation

The CNN model (CNN-AFC) has been trained on the NIST-DB4 fingerprint database and IIIT-D latent fingerprint database. NIST-DB4 consists of 4000 8-bit greyscale fingerprint image of size 512 × 512 pixels, classified into five classes, Arch (*A*), Left Loop (*L*), Right Loop (*R*), Tented Arch (*T*), and Whorl (*W*). IIIT-D Latent Fingerprint database published by Image Analysis and Biometrics lab Indraprastha Institute of Information Technology, Delhi contains 1045 latent fingerprint images from 15 subjects lifted using brush and black powder. The performance of CNN-AFC CNN model in classifying fingerprint and latent fingerprint images is shown in Tables 7.3 and 7.4, respectively. On IIIT-D latent database, CNN-AFC achieved a classification accuracy of 78.2% while as on NIST-DB4 fingerprint database, the classification accuracy mounted up to 92.2%.

7.4.3 Model Improvement Using Transfer Learning

Deep models like Convolutional Neural Network (ConvNet) have large number of parameters that must be learnt before they can be utilized to perform the task of interest. In order to train these models, we require extensively large training dataset with a specific end goal to achieve the desired performance. However, it is relatively subtle to have a dataset of adequate size and due to this reason, it is better not to train the whole deep network from scratch. Rather, it is normal to pretrain a ConvNet on a vast dataset (e.g., ImageNet, which contains around 1.2 million labeled images with 1000 classifications), and after that, use the ConvNet either by fine-tuning the trained model according to the specific task or reuse the learnt weights/parameters in another model for the task of interest. Pretrained deep models designed for one task can be fine-tuned to achieve better accuracy in other similar tasks. This process of reusing or transferring models learnt for one task into another similar task is called transfer learning. *Transfer learning thus refers to extracting the learnt weights from a trained base network (pretrained model) and transferring them to another untrained target network instead of training this target network from scratch.* In this way, features learnt in one network are transferred and reused in other network designed to perform a similar task. Transfer learning can be used in the following ways:

(a) ***ConvNet as fixed feature extractor***: Here, the last fully connected layer (classifier layer) is replaced with a new classifier and this last layer is then trained on new dataset. In this way, the feature extraction layers remain fixed and only the classifier gets fine-tuned. This strategy is best suited when the new dataset is insufficient but similar to the original dataset.
(b) ***Fine-tune whole Model***: Here a pretrained model is used with its last fully connected layer (classifier layer) replaced with a new fully connected layer. The whole network is fine-tuned with a new dataset by continuing backpropagation up to the top layers. In this way, all the weights are fine-tuned for the new task.

Results of transfer learning are reported here using three models: two pretrained models AlexNet and VGG-VD and one model CNN-AFC trained on two data sets. Alexnet and VGG-VD were fine-tuned for the task of fingerprint classification. Their last 1000-way softmax layer was replaced by 5-way softmax layer (to make the models output five probabilities for five fingerprint classes), and then the fully connected layers were retrained on fingerprint images. CNN-AFC was first trained on CIFAR-10 dataset and followed by fine-tuning on NIST-DB4. Out of the three models, VGG-VD produced good results. VGG-VD which pretrained on the large dataset ImageNet (which contains around 1.2 million images with 1000 categories) outperformed the conventional approaches for classification of fingerprints. The classification result of all three models on NIST-DB4 fingerprint database and IIIT-D latent fingerprint database is shown in Table 7.5. Fingerprint classification accuracy of CNN-AFC improved by around 2% from 92.2 to 94.1% with Transfer Learning. The class-wise accuracy of each model is summarized in Tables 7.6, 7.7, and 7.8. Out of the three models, VGG-VD produced best results with 95.1% accuracy followed by CNN-AFC-P with an accuracy of 94.11%. AlexNet produced results with an accuracy of 93.10%.

The performance improvement after using Transfer Learning is presented in Tables 7.9 and 7.10. Fingerprint classification accuracy of CNN-AFC improved by around 2% from 92.2 to 94.1% (see Table 7.10) after Transfer Learning (Fig. 7.13).

Table 7.5 Classification result of the CNN models on NIST-DB4 and IIIT-D latent fingerprint databases

Model	Accuracy (NIST-DB4) (%)	Accuracy (IIIT-D)(%)
VGG-VD	95.01	81.30
CNN-AFC	94.11	79.75
AlexNet	93.10	78.43

Table 7.6 Classification result of VGG-VD on NIST-DB4 fingerprint database

Actual class	Predicted class					Accuracy 95% (%)
	A	*L*	*R*	*T*	*W*	
A	**370**	3	2	25	0	92.5
L	0	**381**	0	13	6	95.3
R	3	0	**382**	11	4	95.5
T	19	5	2	**374**	0	93.5
W	0	5	2	0	**393**	98.25

Table 7.7 Classification result of AlexNet on NIST-DB4 fingerprint database

Actual class	Predicted class					Accuracy 93.1% (%)
	A	*L*	*R*	*T*	*W*	
A	**369**	1	0	30	0	92.25
L	0	**379**	0	17	4	94.75
R	4	0	**367**	25	4	91.75
T	24	11	3	**362**	0	90.5
W	0	7	8	0	**385**	96.25

Table 7.8 Classification result of CNN-AFC on NIST-DB4 fingerprint database

Actual class	Predicted class					Accuracy 94.1% (%)
	A	*L*	*R*	*T*	*W*	
A	**371**	3	2	24	0	92.75
L	2	**383**	1	11	3	95.75
R	7	0	**374**	14	5	93.50
T	23	8	4	**365**	0	91.25
W	1	6	4	0	**389**	97.25

Table 7.9 Overall accuracy of CNN-AFC before and after transfer learning

Model	Accuracy (NIST-DB4) (%)	Accuracy (IIIT-D) (%)
CNN-AFC (before TL)	92.23	78.2
CNN-AFC-P (after TL)	94.11	79.75

Table 7.10 Class-wise accuracy of CNN-AFC before and after Transfer Learning

Actual class	Predicted class					Accuracy before TL 92.2% (%)	Accuracy after TL 94.1% (%)
	A	*L*	*R*	*T*	*W*		
A	361	5	2	32	0	90.25	92.75
L	2	376	2	15	5	94.0	95.75
R	8	0	369	18	5	92.25	93.50
T	34	9	5	352	0	88.0	91.25
W	0	8	6	0	386	96.5	97.25

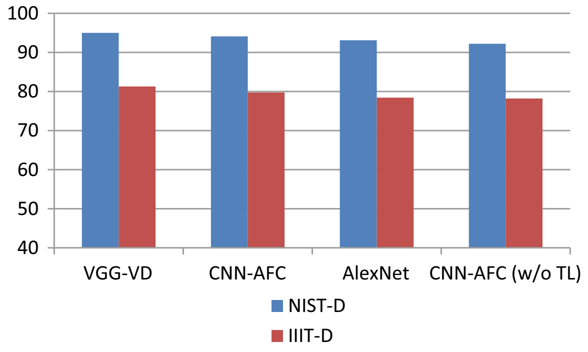


Fig. 7.13 Performance of VGG-VD, AlexNet and CNN-AFC on NIST-DB4 and IIIT-D latent database

Convolutional neural networks tend to learn first-layer features that either resembles Gabor filter or color blobs. This phenomenon is independent of the network architecture and arises not only for different datasets, but also with very different training objectives. Therefore, these top-level features, which are general for all networks and datasets, are called *general* features. On the other hand, the features learned by the last layer of a trained network must be specific to the chosen dataset and task. These features are specific to a particular task and dataset. Thus, these last layer (bottom level) features are called *specific* features. The features from middle layers (layers between top layer and last layer) show the transition from general to specific. For example, if a deep network has n layers with 1 being top layer and n being last layer, then layer 1 will have maximum generality and minimum or zero specificity and layer n will have minimum or zero generality but maximum specificity. Figure 7.14 below shows first-layer features learned by CNN-AFC before and after Transfer Learning. Filters learned by CNN-AFC before transfer learning looks noisy and incomplete while as features learned after TL look refined.

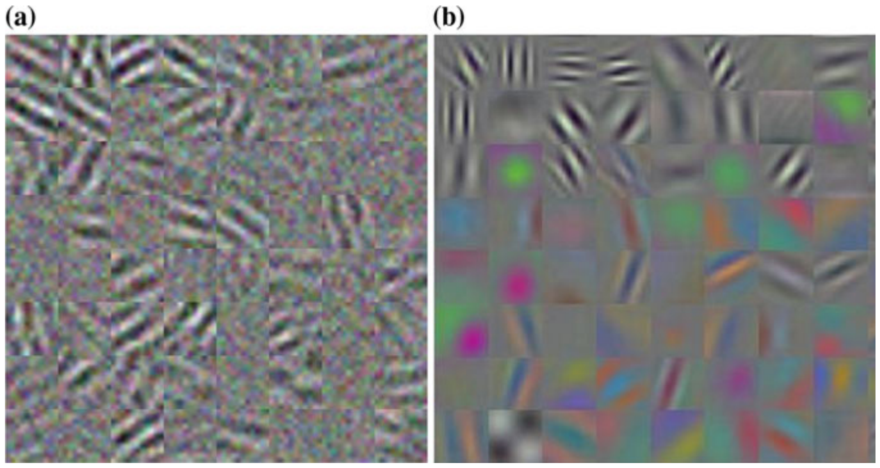


Fig. 7.14 First layer filters learned by CNN-AFC **a** before and, **b** after transfer learning

7.5 Challenges and Future Research Direction

Convolutional Neural Networks can be used at different stages of the fingerprint recognition system. The stages include segmentation, classification, and minutiae extraction. Patch-based segmentation technique using Convolutional Neural Networks has shown promising results on latent fingerprints. However, the patch-based method is computationally expensive as the input image is divided into a number of small patches, and then each patch is fed to a CNN model. An input image with 100 patches will require 100 CNN passes for 100 patches thus making the approach both computationally expensive as well as slow. One way to overcome this issue is to use some region-based technique to divide the input image into regions (regions likely to contain fingerprint) instead of dividing the entire images into equal sized patches. This will confine fingerprint image processing to those areas that are highly likely to contain useful fingerprints. This can reduce the number of CNN passes to a great extent. Another challenge in automatic fingerprint recognition is the extraction of minutiae points from latent fingerprints. A robust CNN-based approach to extract minutiae points from a latent fingerprint image can be of great help to forensic experts.

Bibliography

1. Arshad, I., Raja, G., Khan, A.: Latent fingerprints segmentation: feasibility of using clustering-based automated approach. Arab. J. Sci. Eng. (Springer Science & Business Media BV) **39**(11) (2014)
2. Cao, K., Jain, A.K.: Latent orientation field estimation via convolutional neural network. In: 2015 International Conference on Biometrics (ICB), pp. 349–356. IEEE (2015)
3. Cappelli, R., Maio, D.: The state of the art in fingerprint classification. Autom. Fingerpr. Recognit. Syst., 183–205 (2004)
4. Ezeobiejesi, J., Bhanu, B.: Latent fingerprint image segmentation using fractal dimension features and weighted extreme learning machine ensemble. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 146–154 (2016)
5. Guo, W., Tang, Y.: Latent fingerprint recognition: challenges and advances. In: Biometric Recognition, pp. 208–215. Springer, Cham (2013)
6. Jain, A.K., Hong, L., Pankanti, S., Bolle, R.: An identity-authentication system using fingerprints. Proc. IEEE **85**(9), 1365–1388 (1997)
7. Khan, A.I., Wani, M.A.: Patch-based segmentation of latent fingerprint images using convolutional neural network. Appl. Artif. Intell. **8**, 1–5 (2018)
8. Maltoni, D., Maio, D., Jain, A.K., Prabhakar, S.: Handbook of Fingerprint Recognition. Springer Science & Business Media (2009)
9. Mehtre, B.M., Murthy, N.N., Kapoor, S., Chatterjee, B.: Segmentation of fingerprint images using the directional image. Pattern Recognit. **20**(4), 429–435 (1987)

Chapter 8
Unsupervised Deep Learning in Character Recognition

8.1 Introduction

The recognition of handwritten digits is a well-researched problem and has many applications in real life. The important applications include automatic reading of addresses on postal envelopes, automated form processing, automated processing of handwritten bank cheques, and filled-in forms like questionnaires or money orders. Digit recognition serves as an evaluation task because the problem is well defined and benchmark datasets are easily available.

The first step of this process is to separate the digits by providing strict boundaries that contain these digits. Thus, the main problem that needs to be tackled is to identify the isolated handwritten digits.

8.2 Datasets of Handwritten Digits

A commonly used dataset of handwritten digits is MNIST dataset which is a collection of 70,000 digits. Half of the handwritten digits in the dataset were written by the Census Bureau employees and the rest of the characters were written by high school students. The digits written by Census Bureau employees are much cleaner and easier to recognize than the digits written by students. These digits were stored as 28×28 grayscale images, and were divided into two sets training set of size 60,000 and test set of size 10,000 images. Figure 8.1 shows some of the digits in the training dataset. In this dataset, each digit image is 28 pixels in height and 28 pixels in width, giving a total of 784 pixels for each image. The pixel values, x_i, range from 0 for a completely black pixel to 255 for a completely white pixel. The image can be flattened and represented as a vector $x = (x_1, x_2, \ldots, x_d)^{\mathrm{T}}$ (in this case $d = 784$).

The distribution of pixel values, x_i, have similar patterns for the same numeral. Figure 8.2 shows some images from the training dataset of the handwritten number "6". There are a lower loop and a curve in the top part of the image. For the images

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_8

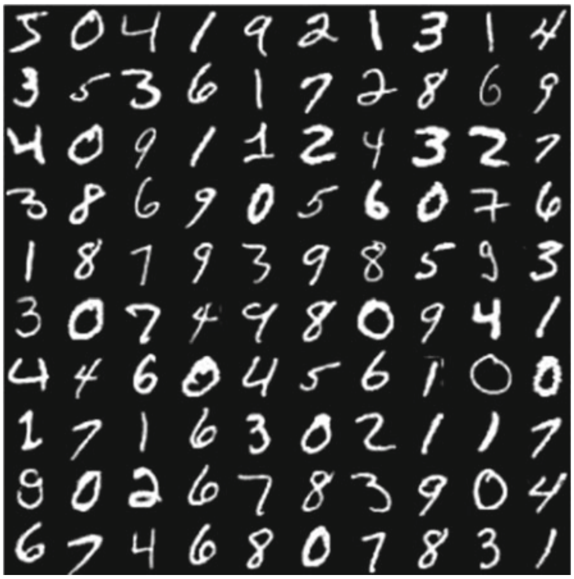


Fig. 8.1 MNIST dataset: First 100 digits in the training dataset

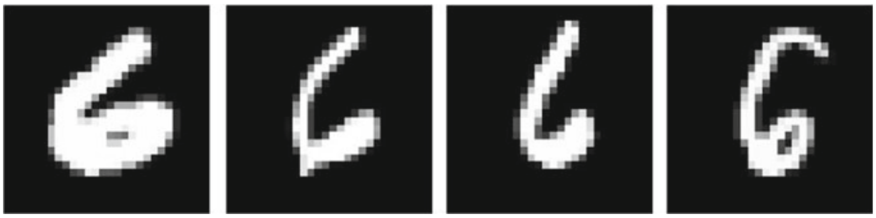


Fig. 8.2 Images of numeral “6” from the training dataset

belonging to the same class, high pixel values are located in similar locations in the vector. The mean image for each numeral is computed as shown in Fig. 8.3 that shows high pixel values locations of various digits.

The number of images of each class present in the training dataset and the testing dataset is shown in Fig. 8.4.

The task is to recognize a numeral from the image and the process includes the following steps: First, $n \times n$ pixel image of a handwritten digit is read. Each pixel of the image is converted into a number between 0 and 1, with 0 as black and 1 as white. It is then saved in a vector denoted as $x = (x_1, x_2, \dots, x_n)$. The label of each image is denoted as $y = y_0, y_1, \dots, y_9$. For example, (0, 0, 0, 0, 0, 1, 0, 0, 0, 0) represents a label of 5. Then different classifier models can be obtained using the training dataset.

The other handwritten digit datasets include USPS and Gisette. The USPS data set consists of 9298 labeled handwritten digit images, each representing a 16×16

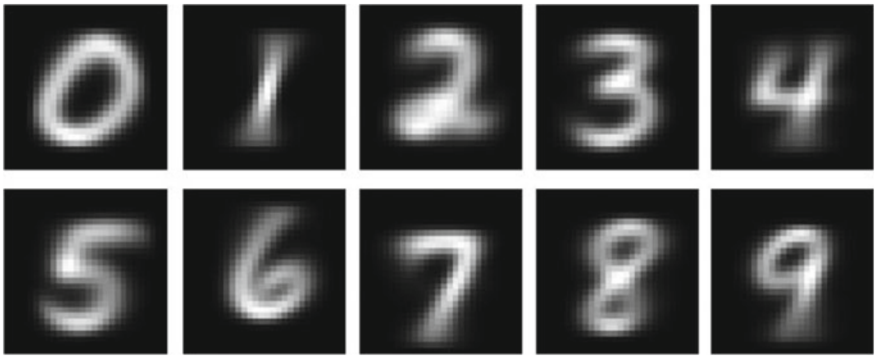


Fig. 8.3 Average of images of each numeral from the training dataset

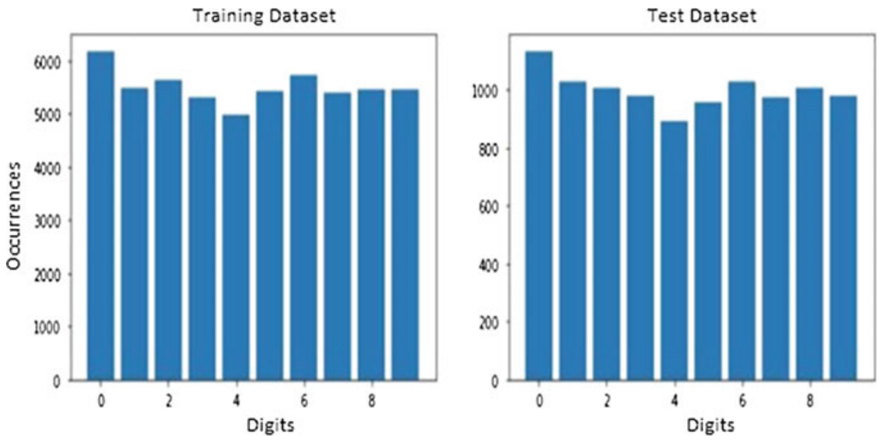


Fig. 8.4 Number of images of each digit in the training and testing datasets

pixel image of one of the 10 digits (0–9). Some training samples from USPS dataset are shown in Fig. 8.5.

The Gisette dataset consists of 6000 labeled handwritten digit images of “4” and “9”. The dataset is divided into 4000 training images and 2000 testing images. The digits are scaled to a uniform size and centered in a 28 by 28 pixel image.

The application of deep learning techniques over the last decade has proven successful in building systems, which are competitive to human performance and which perform better than many traditional AI systems. This chapter will discuss deep learning architectures for handwritten characters.

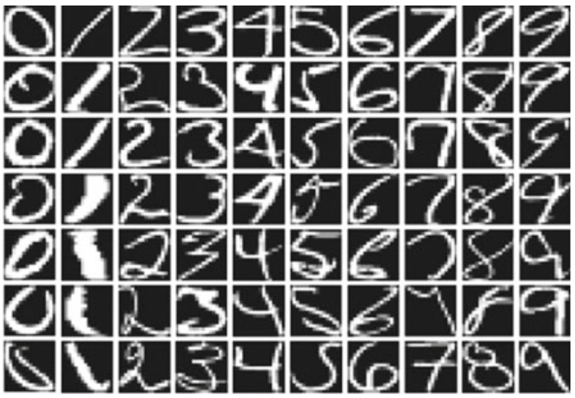


Fig. 8.5 Some training samples from USPS dataset

8.3 Deep Learning Architectures for Character Recognition

The deep learning architecture that is used for recognition of handwritten digits is trained in two phases: During the first phase, the deep network is trained layer by layer in an unsupervised manner. Each layer takes as input the representation produced by the layer before it, with the ultimate goal of discovering more abstract representations as we move up the network. During the second phase, fine-tuning is performed which involves adjusting the parameters of the deep network according to some ultimate task of interest.

8.3.1 Unsupervised Pretraining

Unsupervised pretraining makes its contribution by initializing the parameters of deep networks to sensible values so that they represent the structure of input data in a more meaningful way than the random initialization, thereby yielding significant improvement in the generalization performance. The subsequent fine-tuning phase improves the discrimination ability by slightly modifying the model parameters to adjust the boundaries between classes. Figure 8.6 illustrates a general overview of pretraining and fine-tuning.

For unsupervised pretraining, a stack of RBMs is used which is trained from bottom-up with the representations from each layer used as input to the next RBM. This process is repeated until a desired number of RBM's are trained to create a multilayer model. Finally, the parameters of RBM so obtained are used to initialize the parameters of deep network. A final output layer is added to the deep neural network and the entire network is then fine-tuned using supervised learning algorithm. The weights of the output layer are randomly initialized and fine-tuning updates the weights of all layers by backpropagating the error gradients. The weight update

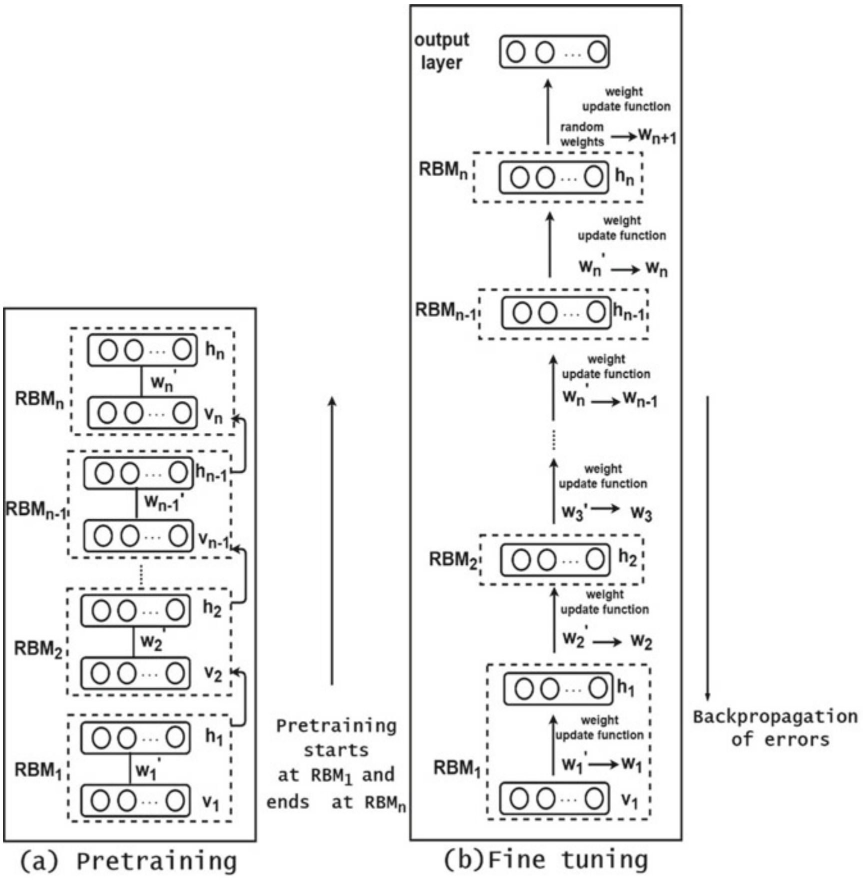


Fig. 8.6 **a** Unsupervised pretraining of *n* layers of deep neural network with restricted Boltzmann machine, **b** supervised fine-tuning of deep neural network

function in Fig. 8.6 represents the weight correction term that is used to update the corresponding weights.

Deep Neural Network (DNN) is constructed in the following manner: The first RBM is trained on the input data by CD algorithm so that the probability distribution represented by it corresponds to the distribution of the training data. After the first RBM learns its weights, the binary states of its hidden units create a second-level representation of data which is used as a visible layer for the next RBM. This process is repeated until a desired number of RBM's are trained to create a multilayer model. Units in the hidden layers learn the complex features from the data that allow the output layer to produce accurate decision boundaries. This stack of trained RBMs is referred to as DBN.

A decision layer is then added to it in order to implement the desired task with training data, creating a DBN-DNN. The structure of DBN-DNN is shown in Fig. 8.7.

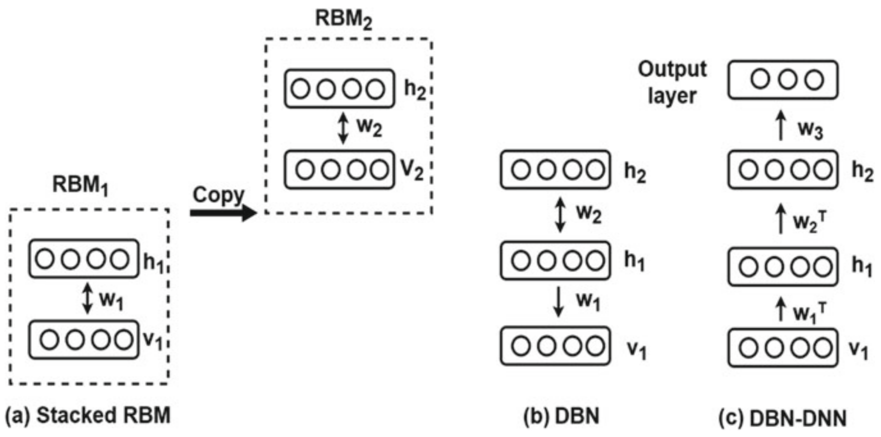


Fig. 8.7 Construction of DBN-DNN

The parameters (weights and biases) of DBN-DNN are then fine-tuned in a supervised manner for better discrimination.

8.3.2 Supervised Fine Tuning

(a) **Fine-Tuning Using BP Algorithm**

A common method that is used for the fine-tuning of a pretrained deep model is the Backpropagation (BP) algorithm. After a deep network has been pretrained through unsupervised learning, the BP algorithm updates the parameters of the model through gradient descent optimization technique. Figure 8.8 shows the fine-tuning done using the standard BP algorithm.

Consider a fully connected feed forward neural network with n_L layers. Let $n \in \{1, \ldots, n_L\}$ be the layers of the network, where n_L is the output layer, $\boldsymbol{s}^{(n)}$ be the input vector into layer n, $\boldsymbol{y}^{(n)}$ be the output vector from layer n, $W^{(n)}$ be the matrix of weights and $b^{(n)}$ vector of bias terms at layer n. The forward flow of activation in the standard BP algorithm (for unit j) can be given as

$$s_j^{(n+1)} = \mathbf{w}_j^{(n+1)}\mathbf{y}^{(n)} + b_j^{(n+1)} \tag{8.1}$$

$$y_j^{(n+1)} = f\left(s_j^{(n+1)}\right) \tag{8.2}$$

where f is sigmoid activation function defined by Eq. (8.3):

Similarly, for the output layer nodes the forward flow of activation can be given as

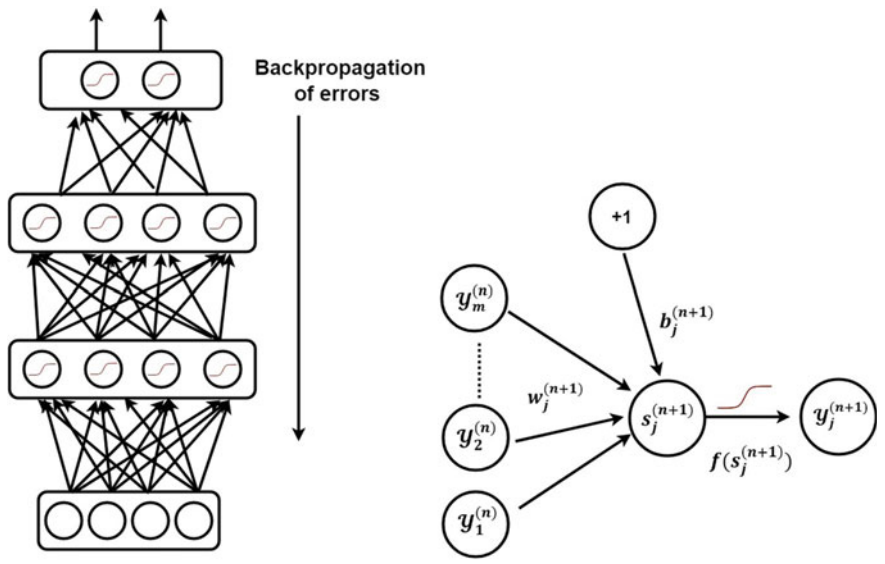


Fig. 8.8 **Left**: Fine-tuning of DBN-DNN using BP, **Right**: basic operation of fine-tuning using BP

$$f\left(s_j^{(n+1)}\right) = 1/1 + exp\left(-s_j^{(n+1)}\right) \tag{8.3}$$

$$s_j^{(n_L)} = \boldsymbol{w}_j^{(n_L)}\mathbf{y}^{(n_L-1)} + b_j^{(n_L)} \tag{8.4}$$

$$\mathcal{Y}_j^{(n_L)} = f\left(s_j^{n_L}\right) \tag{8.5}$$

Once the output from the network is obtained, the error term for the output layer nodes $\delta_j^{(n_L)}$ and the error term for the hidden layer nodes $\delta_j^{(n)}$ can be calculated.

The steps of fine-tuning using BP algorithm are given below.

(i) For a given training example, the activations for layers L_2, L_3 … L_{n_L} are computed.
(ii) For each output unit j in layer n_L, compute the error term as

$$\delta_j^{(n_L)} = (t_j - \mathcal{Y}_j^{(n_L)})f'\left(s_j^{(n_L)}\right) \tag{8.6}$$

(iii) For $n = n_L - 1, n_L - 2, n_L - 3, \ldots 2$

For each node j in layer n, compute the error term as

$$\delta_j^{(n)} = \left(\sum_{i=1} \boldsymbol{w}_{ij}^{(n)}\delta_i^{(n+1)}\right)f'\left(s_j^{(n)}\right) \tag{8.7}$$

(iv) Compute the weight change and bias change as

$$\Delta \boldsymbol{w}_{ji}^{(n)} = \mathcal{Y}_i^{(n)} \delta_j^{(n+1)} \tag{8.8}$$

$$\Delta b_j^{(n)} = \delta_j^{(n+1)} \tag{8.9}$$

(b) **Fine-Tuning Using Dropout-BPAG**

Dropout-BPAG involves integrating adaptive gain backpropagation (BPAG) algorithm with the dropout technique. This algorithm is used in fine-tuning of the constructed DBN-DNN. BPAG algorithm involves adjusting the gain parameter (slope) of the sigmoid function during training, in a manner very similar to that used for adjusting the weights. Varying gain parameter improves the learning efficiency of the trained model and thereby improving the generalization performance. Furthermore, BPAG algorithm overcomes the learning slowdown problem associated with usage of sigmoid units, which gets further aggravated in case of deep networks (Fig. 8.9).

Dropout is a regularization technique that randomly omits some fraction of units in the network to boost neural network accuracy. It prevents coadaptation of neurons, such that each neuron behaves as a reasonable model without relying on other neurons being there. This makes each neuron to be more robust, independently useful and

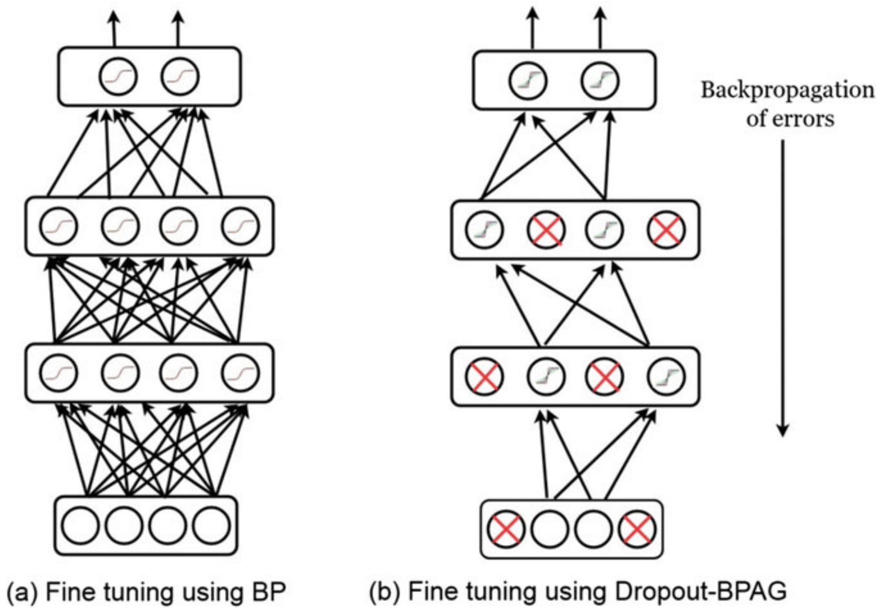


Fig. 8.9 **a** Fine-tuning using BP, **b** fine-tuning using dropout-BPAG, crossed units represent the nodes that have been dropped

pushes it toward creating more meaningful representation instead of relying on others. Neurons are dropped with a probability $q = 1 - p$ and dropping a neuron is equivalent to dropping all its weighted connections. Basically, using dropout involves sampling a subnetwork from the entire network. If a neural network consists of n units, we can have 2^n possible subnetworks.

In the Dropout-BPAG algorithm, for each training case, we sample a subnetwork by dropping out units. We take into consideration the gain parameter of only the neurons that are retained and in a similar manner adapt the gain parameter of only these neurons, as forward and backpropagation for that training case are done only on the subnetwork rather than the entire network. This improves the generalization performance of the model. Figure 8.9 shows fine-tuning done using Dropout-BPAG.

The forward flow of activation in the algorithm (for hidden unit j) of Fig. 8.10 can be given as

$$m^{(n)} \sim \text{Bernoulli}(p) \tag{8.10}$$

$$\tilde{\mathbf{y}}^{(n)} = \boldsymbol{m}^{(n)} * \mathbf{y}^{(n)} \tag{8.11}$$

$$s_j^{(n+1)} = \boldsymbol{w}_j^{(n+1)} \tilde{\mathbf{y}}^{(n)} + b_j^{(n+1)} \tag{8.12}$$

where f is sigmoid activation function defined by

$$\mathcal{Y}_j^{(n+1)} = f\left(s_j^{(n+1)} c_j^{(n+1)}\right) \tag{8.13}$$

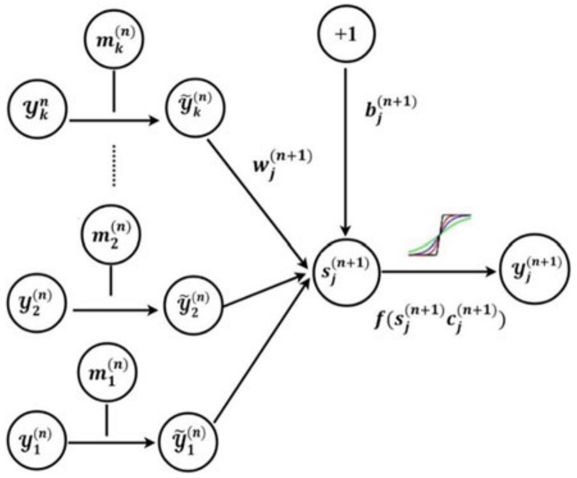


Fig. 8.10 Basic operation of dropout-BPAG

$c_j^{(n+1)}$ is the gain parameter associated with node j of hidden layer $n + 1$ and $\boldsymbol{m}^{(n)}$ is a vector of independent Bernoulli random variables associated with layer n, each of which has the probability p of being 1. The outputs of layer n, $\mathbf{y}^{(n)}$ are then multiplied element wise with the vector $\boldsymbol{m}^{(n)}$, to produce thinned outputs $\tilde{\mathbf{y}}^{(n)}$. The thinned outputs are then passed to a sigmoid function with slope parameter and the outputs so obtained are used as the input to the next layer. This process is repeated at each layer.

In a similar manner, the forward flow of activation (for unit j) for the output layer nodes can be given as

$$\tilde{\mathbf{y}}^{(n_L-1)} = \boldsymbol{m}^{(n_L-1)} * \mathbf{y}^{(n_L-1)} \tag{8.14}$$

$$s_j^{(n_L)} = \boldsymbol{w}_j^{(n_L)} \tilde{\mathbf{y}}^{(n_L-1)} + b_j^{(n_L)} \tag{8.15}$$

$$\mathcal{Y}_j^{(n_L)} = f\left(s_j^{(n_L)} c_j^{(n_L)}\right) \tag{8.16}$$

After the computation of the output from the network, we compute the error term that measures how much a node was responsible for any errors in the output. However, in this algorithm, while calculating the error term, we need to take into consideration the gain parameter of each node at each layer of the subnetwork.

The steps of fine-tuning using Dropout-BPAG are given below.

(i) For a given training example, compute the activations for layers L_2, L_3 … L_{n_L}

(ii) For each output unit j in layer n_L, compute the error term as

$$\delta_j^{(n_L)} = \left(t_j - \mathcal{Y}_j^{(n_L)}\right) f'\left(s_j^{(n_L)}\right)) \tag{8.17}$$

(iii) For $n = n_L - 1, n_L - 2, n_L - 3, \ldots 2$

For each retained node j in layer n, compute the error term as

$$\delta_j^{(n)} = \left(\sum \boldsymbol{w}_{ij}^{(n)} \delta_i^{(n+1)}\right) f'\left(s_j^{(n)}\right) c_i^{(n+1)} \tag{8.18}$$

(iv) Compute the weight, bias, and gain parameter change as

$$\Delta \boldsymbol{w}_{ji}^{(n)} = \mathcal{Y}_i^{(n)} \delta_j^{(n+1)} c_j^{(n+1)} \tag{8.19}$$

$$\Delta b_j^{(n)} = \delta_j^{(n+1)} \tag{8.20}$$

$$c_j^{(n)} = \delta_j^{(n+1)} s_j^{(n+1)} \tag{8.21}$$

(c) **Fine-Tuning Using Dropout-BPGP**

Dropout-BPGP involves integrating the backpropagation with pattern-based gain parameter (BPGP) with the dropout technique. For each training case, a subnetwork is sampled by dropping out units. Only neurons that are retained are considered for performing training on the subnetwork rather than the entire network. This improves the generalization performance of the model. For each training case, a different subnetwork is sampled, with each neuron learning features on its own without relying on the presence of other neurons being there.

The forward flow of activation in the Dropout-BPGP (for hidden unit j) can be given as

$$m^{(n)} \sim \text{Bernoulli}(p) \tag{8.22}$$

$$\tilde{\mathbf{y}}^{(n)} = \boldsymbol{m}^{(n)} * \mathbf{y}^{(n)} \tag{8.23}$$

$$s_j^{(n+1)} = \boldsymbol{w}_j^{(n+1)} \tilde{\mathbf{y}}^{(n)} + b_j^{(n+1)} \tag{8.24}$$

$$y_j^{(n+1)} = f\left(s_j^{(n+1)} c_j^{(n+1)}\right) \tag{8.25}$$

where f is sigmoid activation function defined by

$$f\left(s_j^{(n+1)}\right) = 1/1 + \exp\left(-s_j^{(n+1)} c_j^{(n+1)}\right) \tag{8.26}$$

$c_j^{(n+1)}$ is the gain parameter associated with node j of hidden layer $n + 1$, $\boldsymbol{m}^{(n)}$ is a vector of independent Bernoulli random variables associated with layer n, each of which has the probability p of being 1. The outputs of layer n, $\mathbf{y}^{(n)}$ are then multiplied element wise with the vector $\boldsymbol{m}^{(n)}$, to produce thinned outputs $\tilde{\mathbf{y}}^{(n)}$. The thinned outputs are then passed to a sigmoid function and the outputs so obtained are used as the input to the next layer. This process is repeated at each layer. In a similar manner, the forward flow of activation (for unit j) for the output layer nodes can be given as

$$\tilde{\mathbf{y}}^{(n_L-1)} = \boldsymbol{m}^{(n_L-1)} * \mathbf{y}^{(n_L-1)} \tag{8.27}$$

$$s_j^{(n_L)} = \boldsymbol{w}_j^{(n_L)} \tilde{\mathbf{y}}^{(n_L-1)} + b_j^{(n_L)} \tag{8.28}$$

$$y_j^{(n_L)} = f\left(s_j^{(n_L)} c_j^{(n_L)}\right) \tag{8.29}$$

After the computation of the output from the network, the degree of approximation to the desired output of the output layer is calculated and is used to adjust the value of gain parameter of the nodes in the last hidden layer, while keeping the gain parameter of nodes in the lower hidden layers fixed.

The gain parameter of the nodes in the last hidden layer is then adjusted as

$$c_j^{n_L-1} = \begin{cases} 1/A_p = H/e_p \text{ if } A_p > 1 \\ 0 \qquad\qquad \text{else} \end{cases} \tag{8.30}$$

where A_p represents the approximation degree of output layer defined as $A_p = e_p/H$, H represents the average value of the difference between teacher signals and e_p is computed as $e_p = \max_k \left(\left| \left(t_{kp} - \mathcal{Y}_{kp}^{(n_L)} \right) \right| \right)$, where t_{kp} and $\mathcal{Y}_{kp}$ represent target output and network output for training pattern p; $p \in \{1, \ldots, P\}$ and output node k; $k \in \{1, \ldots, K\}$.

The deep network models are evaluated on the MNIST, USPS, and Gisette handwritten digit datasets. The evaluation is carried out on the basis of classification accuracy, error rate on the test dataset, and root mean squared error.

For MNIST dataset, the deep network consists of four layers, inclusive of the input and output layers, as shown in Fig. 8.11. Fully connected weights are used to link the consecutive layers. The input layer takes input from a 28 × 28 image through a 784-dimensional vector. The successive layers have 1200 hidden variables. The last hidden layer is associated with an output layer consisting of 10 output variables that correspond to 10 class labels, representing a digit. In order to evaluate the effectiveness of the deep architecture, the performance is tested on varying size of MNIST dataset. The MNIST dataset is used to construct four datasets MNIST-20, MNIST-50, MNIST-70, and MNIST-100. These training sets are constructed by randomly choosing training samples of size 20, 50, 70, and 100% from the original dataset.

For USPS dataset, a 256-200-100-10 DBN-DNN is trained as shown in Fig. 8.12.

For Gisette dataset, a four-layer DBN-DNN (5000-200-100-2) is trained as shown in Fig. 8.13.

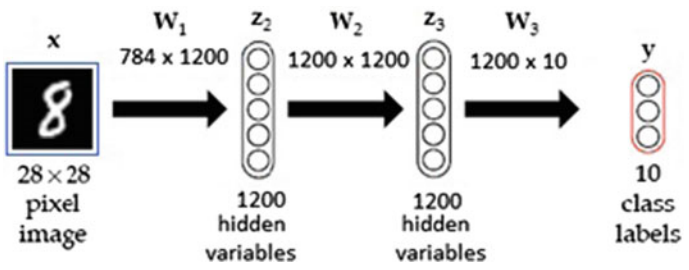


Fig. 8.11 Deep network for MNIST

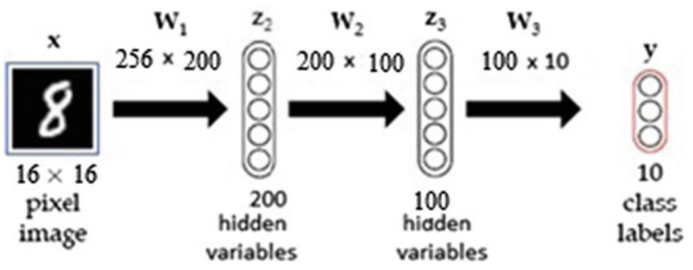


Fig. 8.12 Deep network for USPS dataset

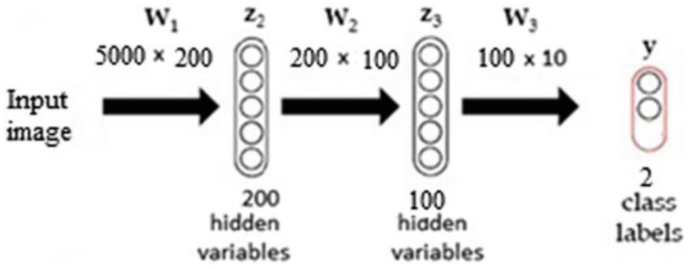


Fig. 8.13 Deep network for Gisette dataset

8.4 Performance Comparison of Deep Learning Architectures

For experimental results, the deep architectures have been trained in two phases, *first phase* involves the construction of DBN-DNN using unsupervised pretraining and the *second phase* involves fine-tuning by using BP, Dropout, Dropout-BPAG, and Dropout-BPGP.

The values of hyper-parameters that are used in the pretraining are: the learning rate in both layers is set to 0.1, initial momentum is set to 0.5 and momentum after the fifth epoch is set to 0.9. The weight penalty l_2 in the pretraining phase is 2×10^{-5}.

The learning rate for the fine-tuning phase is set to 0.1. For the pretraining and fine-tuning phase, the size of the mini batches is set to 100. For dropout, nodes are dropped out at both the input layer as well as at the hidden layer. At the input layer, the input components are retained with the probability of 0.8. While at the hidden layer, the units are retained with probability of 0.5. Each model is trained with 1000 epochs.

The performance is evaluated using the three metrics: testRMSE (root mean squared error), classification accuracy, and the error rate on the test dataset, which are computed as follows:

$$\text{error rate} = N_{\text{inc}}/N, \tag{8.31}$$

$$\text{testRMSE} = \sqrt{1/N \sum_{i=1}^{N} ||t_i - F(x_i)||^2} \tag{8.32}$$

where N_{inc} is the number of missclassified samples, N is the total number of test samples, x_i is the ith test vector, $F(x_i)$ represents the actual output and t_i represents the target output.

(a) *Results on MNIST dataset*

The testRMSE, accuracy, and error rate of the various architectures on different size of MNIST dataset are summarized in Tables 8.1, 8.2, 8.3, and 8.4, respectively (Fig. 8.14).

Table 8.1 Performance of deep architectures on MNIST-20

Deep learning model	Fine-tuning algorithm	testRMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.0941	0.045	95.5
DBN-DNN	BP	0.0677	0.0255	97.45
DBN-DNN	Dropout	0.0599	0.0216	97.84
DBN-DNN	Dropout-BPGP	0.0602	0.021	97.9
DBN_DNN	Dropout-BPAG	0.0602	0.021	97.9

Table 8.2 Performance of deep architectures on MNIST-50

Deep learning model	Fine-tuning algorithm	testRMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.1055	0.0575	94.25
DBN-DNN	BP	0.0628	0.0207	97.93
DBN-DNN	Dropout	0.0496	0.0146	98.54
DBN-DNN	Dropout-BPGP	0.0485	0.0138	98.62
DBN_DNN	Dropout-BPAG	0.0496	0.0146	98.54

Table 8.3 Performance of deep architectures on MNIST-70

Deep learning model	Fine-tuning algorithm	testRMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.1142	0.069	93.1
DBN-DNN	BP	0.0550	0.017	98.3
DBN-DNN	Dropout	0.0446	0.012	98.8
DBN-DNN	Dropout-BPGP	0.0471	0.0127	98.73
DBN_DNN	Dropout-BPAG	0.0412	0.01	99

Table 8.4 Performance of deep architectures on MNIST-100

Deep learning model	Fine-tuning algorithm	test RMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.1261	0.0834	91.66
DBN-DNN	BP	0.0531	0.0149	98.51
DBN-DNN	Dropout	0.0420	0.0107	98.93
DBN-DNN	Dropout-BPGP	0.0422	0.0108	98.92
DBN_DNN	Dropout-BPAG	0.0410	0.0096	99.04

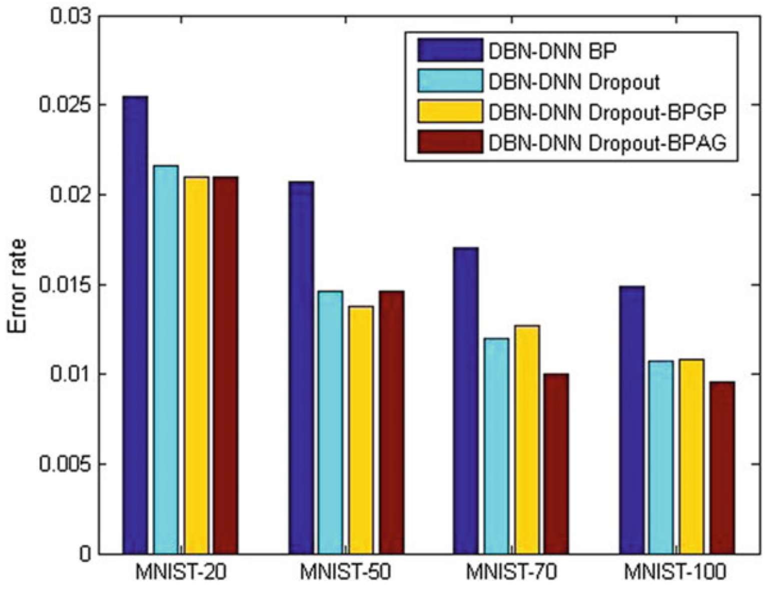


Fig. 8.14 Error rate of deep architectures on MNIST

(b) *Results on USPS dataset*

The testRMSE, accuracy, and error rate of deep architectures on USPS dataset is summarized in Table 8.5.

Table 8.5 Performance of deep architectures on USPS

Deep learning model	Fine-tuning algorithm	testRMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.1222	0.0871	91.29
DBN-DNN	BP	0.0952	0.0538	94.62
DBN-DNN	Dropout	0.0951	0.0523	94.77
DBN-DNN	Dropout-BPGP	0.0950	0.0508	94.92
DBN_DNN	Dropout-BPAG	0.0927	0.0503	94.97

(c) *Results on Gisette*

The testRMSE, accuracy, and error rate of deep architectures on Gisette dataset is summarized in the Table 8.6 (Fig. 8.15).

Table 8.6 Performance of deep architectures on Gisette

Deep learning model	Fine-tuning algorithm	testRMSE	Error rate	Accuracy (%)
DBN-DNN	None	0.1655	0.0355	96.45
DBN-DNN	BP	0.1329	0.02	98
DBN-DNN	Dropout	0.1346	0.0195	98.05
DBN-DNN	Dropout-BPGP	0.1253	0.0175	98.25
DBN_DNN	Dropout-BPAG	0.1277	0.0169	98.31

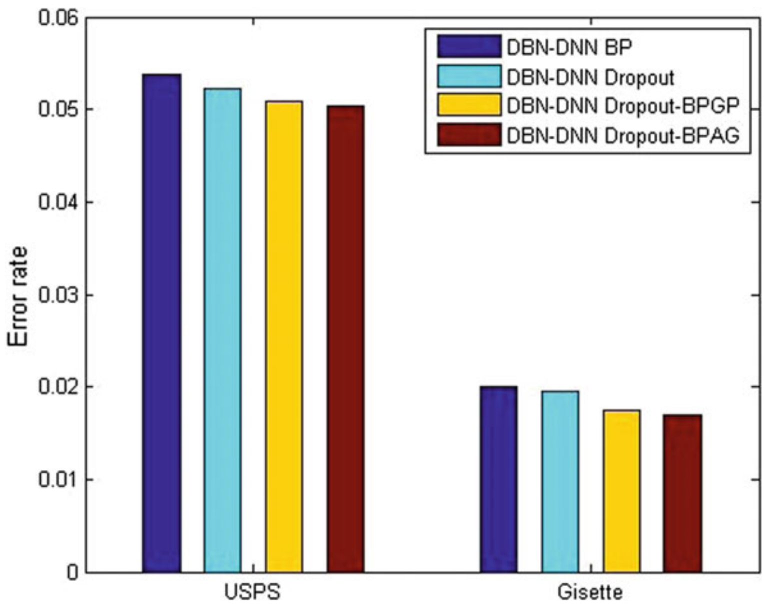


Fig. 8.15 Error rate of deep architectures on USPS and Gisette

8.5 Challenges and Future Research Direction

Unsupervised pretraining followed by supervised fine-tuning presents promising results in handwritten digit recognition. There are a number of areas that are characterized by large volumes of unlabeled data where unsupervised deep architectures can be employed. However, one of the challenges is to determine if the higher layers have significantly adequate information about the original data that is presented at the bottom layers. Another challenge is to determine robust designs of deep learning architectures and changes required in the existing architectures that allow maximum information about the original data to be propagated through to higher layers.

For applications where both labeled and unlabeled data is available, hybrid architecture that makes simultaneous use of supervised and unsupervised deep learning architectures can be explored.

Bibliography

LeCun, Y., Bottou, L., Orr, G.B., Müller, K.R.: Efficient backprop. In: Neural networks: tricks of the trade, pp. 9–50. Springer, Berlin, Heidelberg (1998)

Nawi, N.M., Hamid, N.A., Ransing, R.S., Ghazali, R., Salleh, M.N.M.: Enhancing back propagation neural network algorithm with adaptive gain on classification problems. Int. J. Database Theory Appl. **4**(2) (2011)

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)

Wang, S., Manning, C.: Fast dropout training. In: International Conference on Machine Learning, pp. 118–126 (2013, Feb)

Wang, X., Tang, Z., Tamura, H., Ishii, M., Sun, W.D.: An improved backpropagation algorithm to avoid the local minima problem. Neurocomputing **56**, 455–460 (2004)

Wani, M.A., Afzal, S.: Optimization of deep network models through fine tuning. Int. J. Intell. Comput. Cybern. **11**(3), 386–403 (2018a)

Wani, M.A., Afzal, S.: Gain parameter and dropout-based fine tuning of deep networks. Int. J. Intell. Inf. Database Syst. **11**(4), 236–254 (2018b)