EAI/Springer Innovations in Communication and Computing

Kolla Bhanu Prakash
Ramani Kannan
S. Albert Alexander
G. R. Kanagachidambaresan *Editors*

Advanced Deep Learning for Engineers and Scientists

A Practical Approach

EAI
RESEARCH MEETS INNOVATION

Springer

EAI/Springer Innovations in Communication and Computing

Series Editor

Imrich Chlamtac, European Alliance for Innovation, Ghent, Belgium

Editor's Note

The impact of information technologies is creating a new world yet not fully understood. The extent and speed of economic, life style and social changes already perceived in everyday life is hard to estimate without understanding the technological driving forces behind it. This series presents contributed volumes featuring the latest research and development in the various information engineering technologies that play a key role in this process.

The range of topics, focusing primarily on communications and computing engineering include, but are not limited to, wireless networks; mobile communication; design and learning; gaming; interaction; e-health and pervasive healthcare; energy management; smart grids; internet of things; cognitive radio networks; computation; cloud computing; ubiquitous connectivity, and in mode general smart living, smart cities, Internet of Things and more. The series publishes a combination of expanded papers selected from hosted and sponsored European Alliance for Innovation (EAI) conferences that present cutting edge, global research as well as provide new perspectives on traditional related engineering fields. This content, complemented with open calls for contribution of book titles and individual chapters, together maintain Springer's and EAI's high standards of academic excellence. The audience for the books consists of researchers, industry professionals, advanced level students as well as practitioners in related fields of activity include information and communication specialists, security experts, economists, urban planners, doctors, and in general representatives in all those walks of life affected ad contributing to the information revolution.
Indexing: This series is indexed in Scopus, Ei Compendex, and zbMATH.

About EAI

EAI is a grassroots member organization initiated through cooperation between businesses, public, private and government organizations to address the global challenges of Europe's future competitiveness and link the European Research community with its counterparts around the globe. EAI reaches out to hundreds of thousands of individual subscribers on all continents and collaborates with an institutional member base including Fortune 500 companies, government organizations, and educational institutions, provide a free research and innovation platform.

Through its open free membership model EAI promotes a new research and innovation culture based on collaboration, connectivity and recognition of excellence by community.

More information about this series at http://www.springer.com/series/15427

Kolla Bhanu Prakash • Ramani Kannan
S. Albert Alexander
G. R. Kanagachidambaresan
Editors

Advanced Deep Learning for Engineers and Scientists

A Practical Approach

Springer

EAI
RESEARCH MEETS INNOVATION

Editors

Kolla Bhanu Prakash
KL Deemed to be University
Vijayawada, AP, India

S. Albert Alexander
Kongu Engineering College
Perundurai, Tamil Nadu, India

Ramani Kannan
Electrical & Electronics Engineering
Universiti Teknologi Petronas
Seri Iskandar, Perak, Malaysia

G. R. Kanagachidambaresan
Department of CSE
Vel Tech Rangarajan Dr Sagunthala R&D
Institute of Science and Technology
Chennai, Tamil Nadu, India

ISSN 2522-8595 ISSN 2522-8609 (electronic)
EAI/Springer Innovations in Communication and Computing
ISBN 978-3-030-66518-0 ISBN 978-3-030-66519-7 (eBook)
https://doi.org/10.1007/978-3-030-66519-7

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2021, Corrected Publication 2021
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To our family, students, scholars, and dear friends

Preface

Deep learning is the latest buzzword to understand and utilize new sensor technology and is a primary tool for smart city development. New adaptation and infrastructure development in developing countries are mainly influenced by computer technologies such as deep learning and machine learning approaches. Deep learning methods can cover wider engineering disciplines with various real-world applications such as speech recognition, computer vision, and smart computing scenarios. These methods require broad domain knowledge of various subjects; for example, advanced driver-assistance system (ADAS) design requires the knowledge of maths, physics, computer vision (CV), mechanics, and sensor technology. Various deep learning concepts about healthcare, smart systems, recommendation systems, etc., with real-time discussions are covered in this book. Deep learning is capable of working with unstructured data that requires more time for humans to extract the relevant information. Industry 4.0 standards have perceived the power of deep learning methodology and are utilizing the artificial intelligence (AI) systems for cyber-physical system (CPS). Deep learning is a subset of machine learning in artificial intelligence (AI) that utilizes an artificial neural network (ANN) to operate CPS. The current pandemic situation has attracted several researchers to study the design of healthcare-oriented systems using machine learning algorithms. The outbreak of vector-borne diseases – such as chikungunya, malaria, and dengue – predicted using machine learning as well as its spread in the Indian subcontinent is elucidated in detail. During this covid-19 period, researchers focus on ways for the identification of increase in cholesterol level as well as for contactless sensing and diagnosis. The prediction of eukaryotic plasma cholesterol from human G protein-coupled receptors (GPCRs) is studied through K-nearest neighbor (KNN) and support vector machine (SVM) approach. In many cases, a hybrid approach is required to balance the drawbacks of each approach and to derive a potential application for better prediction. In this book, the hybrid KNN–SVM approach is shown to examine the pattern of cholesterol level, and the obtained data is validated with experimental studies. The detection of early-stage diabetic retinopathy using convolutional neural network (CNN) approach is also addressed. Due to its novelty and accuracy, the retinal scan comparison is performed with the data set to assess the present eye condition of a person. The most widely used deep learning techniques, such as CNN, recurrent neural networks (RNN), Tensorflow, Python tool, deep convolutional generative adversarial network (DCGAN), Auto-encoders, long short-term memory (LSTM), and gated recurrent unit (GRU), are discussed. The identification of image and text using these algorithms, their working perspectives, and their contextual speed are explained for readers. Deep learning in bioinformatics enables to identify hidden information and make correct predictions. Omics data analysis, protein structure prediction, and biomedical image processing are explained with real-time data set. Character recognition and Opencv are mostly used in CPS for automation and machine-to-machine interaction. Both Opencv and character recognition algorithms are capable of operating in single-board computers (SBCs) and have a wide application in the current Industry 4.0 era. CNN-based document analysis, document recognition, scene-text classification, and localizations were discussed with real-time results, and the comparison is also done with standard benchmark algorithms. This book provides an overview of deep learning and machine learning techniques for better prediction and smart computing environments.

Vijayawada, AP, India	Kolla Bhanu Prakash
Seri Iskandar, Perak, Malaysia	Ramani Kannan
Perundurai, Tamil Nadu, India	S. Albert Alexander
Chennai, Tamil Nadu, India	G. R. Kanagachidambaresan

Acknowledgments

We thank the Almighty, our parents, and our spouses for the endless support, guidance, and love through all our life stages. We are thankful to our beloved family members for standing beside us throughout our career, to move our career forward through editing this book.

We specially thank Sri. Koneru Satyanarayana, President, KL University, India, for his continuous support and encouragement for the preparation of this book. We dedicate this book to him, our collaborators, and our family members.

Our great thanks to our students, who have put in their time and effort to support and contribute in some manner. We express our gratitude toward all who supported, shared, talked things over, read, wrote, offered comments, allowed us to quote their remarks, and assisted in editing, proofreading, and designing through this book journey.

We believe that the team of authors provides the perfect blend of knowledge and skills that went into authoring this book. We thank each of the authors for devoting their time, patience, perseverance, and effort toward this book; we think that it will be a great asset to all researchers in this field!

We are grateful to the Springer Publishing team, who showed us the ropes to write this book. Without their support and knowledge, we would not have ventured into starting and creating this book. Their trust in us and their guidance – providing the necessary time and resources – gave us the freedom to manage this book.

Last but not least, we thank our readers, who gave us their trust – we hope our work inspires and guides them.

Kolla Bhanu Prakash
Ramani Kannan
S. Albert Alexander
G. R. Kanagachidambaresan

About the Book

This book contains 11 important contributions addressing current real-time problems. This pandemic period has attracted several researchers on building many healthcare-related projects and algorithms on disease prediction and diagnosis approaches. Deep learning is able to work with unstructured input which requires more time for humans to extract the relevant information. Industry 4.0 CPS standards have perceived the power of deep learning methodology and are utilizing the AI systems for industrial machines and smart city development. This book will be a useful resource to recognize and understand the potential of deep learning for researchers and engineers working on smart city projects and healthcare domain.

Contents

About the Editors

Kolla Bhanu Prakash is working as a professor and research group head in CSE Department, KL University, Vijayawada, Andhra Pradesh, India. He received his MSc and MPhil in Physics from Acharya Nagarjuna University, Guntur, India, and ME and PhD in Computer Science Engineering from Sathyabama University, Chennai, India. He has more than 15 years of experience working in academia, research, teaching, and academic administration. His research interests include artificial intelligence, deep learning, data science, smart grids, cyberphysical systems, cryptocurrency, blockchain technology, and image processing. He is IEEE Senior Member and Fellow-ISRD, Treasurer – ACM Amaravathi Chapter, India, LMISTE, MIAENG, and SMIRED. He has reviewed more than 125 peer-reviewed journals, which are indexed in Publons. He is the editor for six books in Elsevier, CRC Press, Springer, Wiley, and De Gruyter publishers. He has published 65 research papers, 5 patents, 5 books, and 4 accepted. His Scopus H-index is 12. He is a frequent editorial board member and TPC member in flagship conferences and refereed journals. He is a reviewer for Inderscience Publishers and journals such as *IEEE Access*, *Springer Nature*, *Applied Soft Computing Journal* (Elsevier), *Wireless Networks*, *IET*, *KSII*, and *IEEE Computer Society*. He is a series editor of *Next Generation Computing & Communication Engineering* (Wiley); under this series, he had signed agreements for five books.

Ramani Kannan is a senior lecturer in Universiti Teknologi PETRONAS, Malaysia. He received his BE from Bharathiyar University, India. Later, he completed his ME and PhD in Power Electronics and Drives from Anna University. He holds more than 120 publications in reputed international and national journals and conferences. He is a chartered engineer (CEng, UK), active senior member in IEEE (USA), and members of IE(I), IET(UK), ISTE(I), and Institute of Advanced Engineering and Science (IAENS). He has received several awards such as Career Award for Young Teacher from AICTE, India, in 2012; Young Scientist Award in power electronics and Drives, 2015; Highest Research publication Award in 2017; Award for Outstanding Performance, Service and Dedication in 2019 at UTP, Malaysia; Outstanding Researcher Award in UTP Q Day 2019; Best Presenter

Award, IEEE-CENCON 2019 international conference in Indonesia. He is actively serving as a secretary for IEEE Power Electronics Society, Malaysia, since 2020.

He is the editor-in-chief of *Asian Scientific Research* (2011–2018) and regional editor of *International Journal of Computer Aided Engineering and Technology*, Inderscience Publisher, UK, since 2015. He is an associate editor for journals *IEEE Access* since 2018 and *Advanced Materials Science and Technology since 2020.* He is serving as a guest editor for Elsevier, Inderscience, IGI Global, CRC, Taylor & Francis, Bentham Science, and IJPAM. His research interests involve power electronics, inverters, modeling of induction motor, artificial intelligence, machine learning, and optimization techniques.

S. Albert Alexander is a postdoctoral research fellow from Northeastern University, Boston, MA, USA. He is the recipient of prestigious Raman Research Fellowship from the University Grants Commission (Government of India). His research focuses on fault diagnostic systems for solar energy conversion systems and smart grids. He has 14 years of academic and research experience. He has published 25 technical papers in international and national journals (including IEEE Transactions, IET, Elsevier, Taylor & Francis, and Wiley) and presented 23 papers at national and international conferences.

He has completed four projects funded by the Government of India, and one multilateral project is in progress with the overall grant amount of Rs.103 lakhs. His PhD work on power quality earned him a National Award from ISTE, and he has received 25 awards for his meritorious academic and research career (such as Young Engineers Award from IE(I) and Young Scientist Award from SPRERI, Gujarat). He has also received the National Teaching Innovator Award from MHRD (Government of India). He is an approved "Margadarshak" from AICTE (Government of India). He is the approved Mentor for Change under Atal Innovation Mission.

He has guided 32 graduate and postgraduate projects. He is presently guiding seven research scholars and one completed his PhD under his guidance. He is a member and holds prestigious positions in various national and international forums (such as a senior member of IEEE and vice president of Energy Conservation Society, India). He has been an invited speaker in 215 programs both in India and the USA. He has organized 12 events, including faculty development programs, workshops, and seminars.

He has completed his graduate program in Electrical and Electronics Engineering from Bharathiar University and his postgraduate program from Anna University, India. Presently, he is working as an associate professor in the Department of EEE, Kongu Engineering College, and also doing research work in artificial intelligent controllers, smart healthcare systems, smart grids, solar PV, and power quality improvement techniques. He has authored the following books in his areas of interest: (1) *Power Electronic Converters for Solar PV Systems*, Elsevier; (2) *Computational Paradigm Techniques for Enhancing Electric Power Quality*, CRC Press; (3) *Basic Electrical, Electronics and Measurement Engineering*, Anuradha Publishers; and (4) *Special Electrical Machines*, Anuradha Publishers.

G. R. Kanagachidambaresan received his BE in Electrical and Electronics Engineering from Anna University in 2010 and ME in Pervasive Computing Technologies from Anna University in 2012. He has completed his PhD (Wireless Body Area Networks and Healthcare) from Anna University, Chennai, in 2017. He is currently an associate professor in the Department of CSE in Veltech Rangarajan Dr Sagunthala R&D Institute of Science and Technology. His research areas of interest include body sensor network and fault-tolerant wireless sensor network. He has published several articles in reputed journals and has undertaken several consultancy activities for leading MNC companies. He is serving as editorial review board members for peer-reviewed journals and has also guest-edited several special issue volumes and books in Springer. He is presently working on several government-sponsored research projects of ISRO, DBT, and DST.

Introduction to Deep Learning

R. Indrakumari, T. Poongodi, and Kiran Singh

1 Introduction

The human brain is the incredible organ that dictates the signals received from sound, sight, smell, touch, and taste. The brain stores emotions, experiences, memories, and even dreams. The brain takes decisions and solves many problems that even the powerful supercomputers lack [1]. Based on this, researchers are dreamed of constructing intelligent machines like the brain. Later researchers invented robots to assist human activities, automatic disease detection microscopes, and self-driving cars. These inventions still required human interventions to do some computational problems. To tackle this problem, researchers want to build a machine that can learn by themselves and solve more complex problems in the speed of the human brain. These necessities pave the way to the most active field of artificial machine intelligence called deep learning.

2 Neurons

The basic unit of the human brain is the neurons. Very small portions of the brain, about the size of wheat, have over 10,000 neurons with more than 6000 connections with other neurons [2]. The information perceived by the brain is captured by the neurons, and the same is passed from a neuron to others for processing, and the final result is sent to other cells. It is depicted in Fig. 1. Dendrites are an antenna-like structure in the neurons that receives the inputs. Based on the frequency of usage,

R. Indrakumari (✉) · T. Poongodi · K. Singh
School of Computing Science and Engineering, Galgotias University,
Greater Noida, Uttar Pradesh, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_1

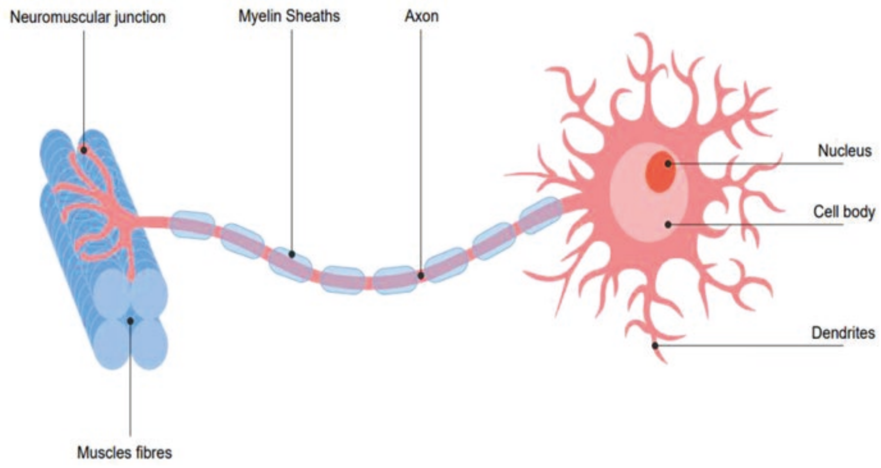


Fig. 1 Biological neuron's structure

the inputs are classified into strengthened and weakened. The connection strength estimates the involvement of the input pertaining to the neuron's output. The input signals are weighted by the connection strength and summed collectively in the cell body. The calculated sum takes the form of a new signal, and it is thriven along the cell's axon to reach the destination neurons.

In 1943, Warren S. McCulloch and Walter H. Pitts [3] concentrated on the functional understanding of the neurons that exist in the human brain and created a computer-based artificial model as shown in Fig. 2.

As in the biological neurons, the artificial neuron receives inputs x_1, x_2, $x_3 \ldots . x_n$, and respectively input is multiplied by particular weights w_1, w_2, $w_3, \ldots . w_n$, and the calculated sum is considered to make the *logit* of the neuron:

$$Z = \sum_{i=0}^{n} w_i x_i \tag{1}$$

Some logit may include a constant value called the bias. Finally, the logit is passed through a function f to make the desired output y = f (z).

3 History of Deep Learning

The history of deep learning started in the early 1940s when Warren McCulloch and Walter Pitts developed a computer model focusing on the human neural system. They applied mathematics and algorithms and called it "threshold logic" to imitate the thinking process. Deep learning is a subsequent derivative of machine learning that applies algorithms, processes the data, and develops abstractions. Various

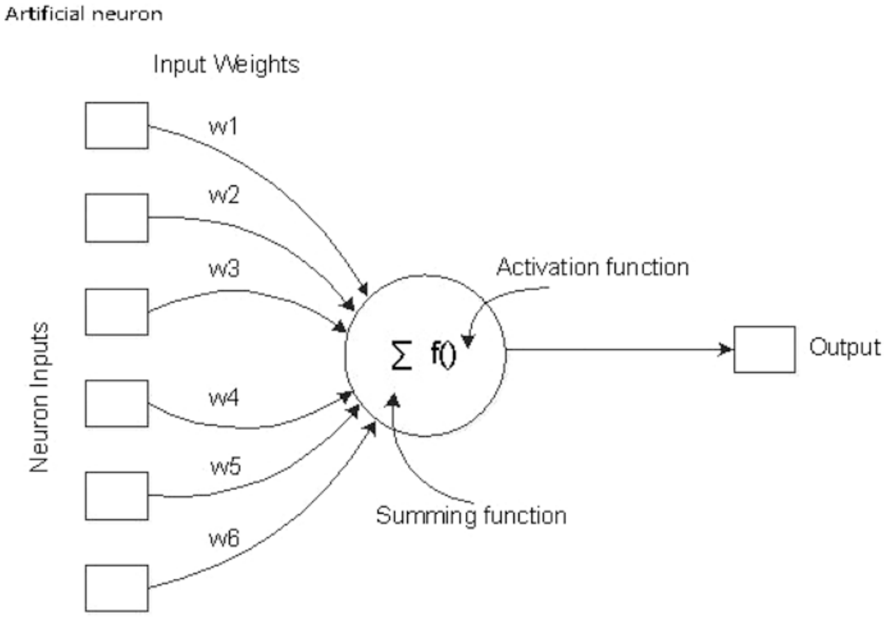


Fig. 2 Neuron in an artificial neural net

algorithms are applied to process data, to recognize objects and human speech. The output of the former layer is provided as the input to the next layer.

In 1960 Henry J. Kelley has started to develop the Backpropagation Model and was extended by Stuart Dreyfus in 1962. The early version of Backpropagation was not so efficient and clumsy. Following this, in 1965, Valentin Grigor'evich Lapa has proposed cybernetics and forecasting techniques, and Alexey Grigoryevich Ivakhnenko has proposed the data handling methodology using polynomial activation functions. The best feature chosen statistically is forwarded to the next layer manually.

Kunihiko Fukushima has developed the first convolutional neural networks with multiple pooling and convolutional layers. Later in 1979, he developed neocognitron, a multilayered and hierarchical artificial neural network design that can recognize visual patterns. Neocognitron is said to be the best model at that time as it uses new learning methods with top-down connections. It contains the selective attention model which recognizes the individual patterns. The developed neocognitron can be able to identify the unknown and missing information with a concept called inference.

In the late 1970s, Seppo Linnainmaa wrote a Fortran code for backpropagation. In 1985, Williams and Hinton studied that backpropagation can provide "interesting" distribution representations. Yann LeCun combined backpropagation with convolutional neural networks and showed the first practical demonstration to read "handwritten" digits at Bell Labs in 1989. Later many optimistic researchers

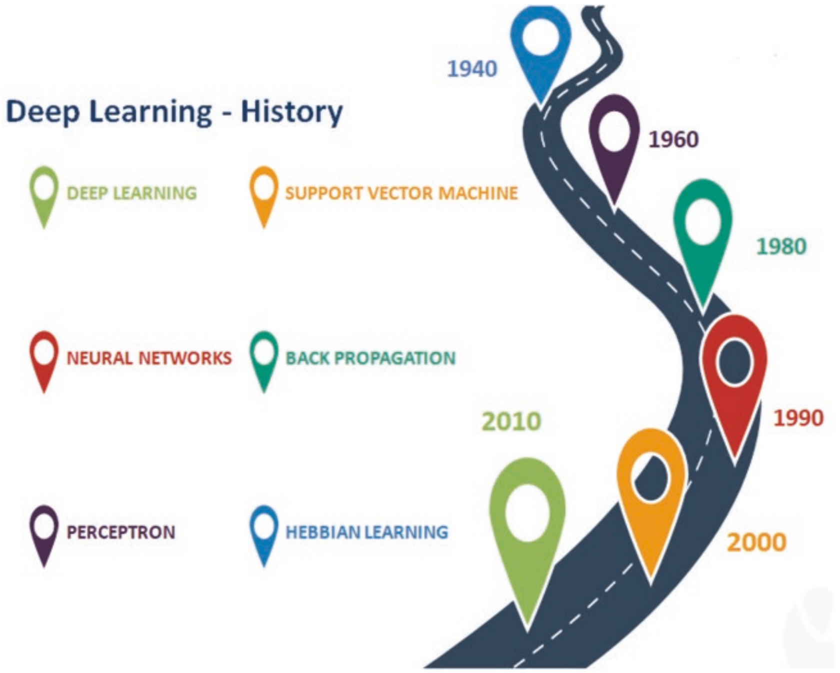


Fig. 3 Roadmap of deep learning history

exaggerated artificial intelligence; notably in 1995, Dana Cortes and Vladimir Vapnik have proposed a model to map and identify similar data, called the support vector machine. In 1997, Sepp Hochreiter and Juergen Schmidhuber have proposed long short-term memory (LSTM) for recurrent neural networks (Fig. 3).

The new era for deep learning began in 1999 as it is the evolution of graphics processing units (GPUs). In 2000, the vanishing gradient problem is identified which paved the way for the development of long short-term memory. Fei-Fei Li an AI expert assembled ImageNet which can process more than 14 million labeled images. During 2011 and 2012, AlexNet a convolutional neural network won many international competitions. In 2012, Google Brain announced a project called The Cat Experiment, which overcomes the limitations of unsupervised learning. At present, the evolution of artificial intelligence and the processing of big data are dependent on deep learning.

4 Feed-Forward Neural Networks

The neurons in the human brain are arranged as layered structure, and even most of the human intelligence part in the brain, the cerebral cortex, is of six layers [4]. The perceived information travels from layer to another layer until obtaining the conceptual understanding from the sensory input.

In Fig. 4, a three-layer perceptron is shown with the hidden layer that contains neurons with nonlinear activation functions. Arbitrarily complex decision and computation of any likelihood function can be easily done by a three-layer perceptron.

From Fig. 4, it is noted that the connection traverses from the low-level layer to the high-level layer and there are no communications among neurons which exist in the same layer as well from the higher to the lower level. Hence these setup is called the feed-forward networks. The middle layer in Fig. 4 is the hidden layer where the magic happens when the neural network tries to solve complex problems. Every layer in Fig. 4 has an equal number of neurons, which is not mandatory. The input and output are represented as vectors. Linear neurons are represented by a linear function in the form of $f_z = a_z + b$. Linear neurons are easy to compute but restricted with limitations. A feed-forward network with only linear neurons contains no hidden layer which enables the users to get vital features from the input layer. In practice, there are three possible types of neurons, namely, sigmoid neuron, tanh neurons, and ReLU neurons, that dumped the nonlinearity concept. The sigmoid neurons use the function

$$f = \frac{1}{1 + e^{-z}} \tag{2}$$

The above equation represents that when the value of logit is actually small, then the output is very close to 0, and it is 1 when the value of logistic is very large. Between the values 0 and 1, the neuron takes the shape of S as shown in Fig. 4.

Based on the types of connections the neural network architecture is categorized into "recurrent neural networks" in which there exists a synaptic connection from output to the input whereas in "feed-forward neural networks" there exists a feedback operation from output to inputs. Neural networks are constructed as either single layers or multilayer.

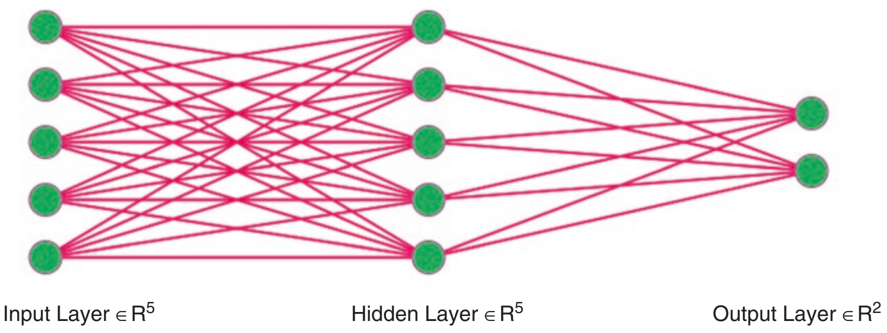


Fig. 4 Three-layer perceptron network with continuous inputs, two output, and two hidden layers

4.1 Backpropagation

Backpropagation is the heart of neural network training which fine-tunes the weights of neural net obtained in the previous epoch. It was developed in 1970, and researchers fully appreciated it after 1986 when David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper describing that backpropagation works faster and provides solutions for previously unsolved problems. Backpropagation is a kind of supervised learning method for multilayer artificial neural networks (ANNs) with applications ranging from classification, pattern recognition, medical diagnostics, etc. The backpropagation algorithm made the multilayer perceptron networks occupy a place in the neural network's research toolbox. The multilayer perceptron is perceived as a feed-forward network with more than one layer of nodes between the input and output nodes. It updates the synaptic weights by propagating a gradient vector back to the input in which the elements are defined as the derivative of an error measure for a parameter. The error signals are the significant difference between the actual and the desired outputs.

The backpropagation algorithms are considered as a generalized view of the least-mean-square (LMS) algorithm that consists of a forward pass and a backward pass. The backpropagation computes specifically all the partial derivatives $\frac{\partial f}{\partial w_i}$ where w_i is the ith parameter and f is the output.

Consider a multilayer feed-forward neural network as shown in Fig. 2. Let us assume a neuron i is present in the output layer and the error signal for n^{th} iteration is given by the equation

$$ei(m) = di - yi(m) \tag{3}$$

where d_i is the desired output for neuron i and $y\,j\,(m)$ is the actual output for neuron i, computed using the current weights of the network at iteration m.

Equation 2 represents the instant error energy value y for the neuron i as

$$\varepsilon_i(m) = \frac{1}{2} e_i^2(m) \tag{4}$$

The instantaneous value $\varepsilon_i(m)$ is the sum of all $\varepsilon_i(m)$ for all neurons in the output layer as represented in Eq. 3

$$\varepsilon_i(m) = \frac{1}{2} \sum_{i \varepsilon S} e_i^2(m) \tag{5}$$

where S is the set of all neurons present in the output layer. For consideration, suppose a training set contains N patterns, and the average square energy for the network is given by Eq. 4:

$$\varepsilon_{avg} = \frac{1}{N}\sum_{n=1}^{N}\varepsilon(m) \tag{6}$$

The modes of backpropagation algorithms are (a) batch mode and (b) sequential mode. In the batch mode, the weight updates are done after an epoch is completed. In contrast to this, the sequential mode or stochastic mode updates are performed after the presentation of each training example. The following equation gives the output expression for the neuron i

$$y_i(m) = f\left[\sum_{i=0}^{n} w_{ij}(m)\, y_i(m)\right] \tag{7}$$

where n represents the total number of inputs to the neuron i from the previous layer and f is the activation function used in the neuron i.

The updated weight to be applied to the weights of the neuron i is directly proportional to the partial derivative of the instantaneous error energy $\varepsilon(n)$ for the corresponding weight, and it is represented as

$$\frac{\partial \varepsilon(m)}{\partial w_{ij}(m)} \tag{8}$$

Using the chain rule of calculus, it is expressed as

$$\frac{\partial \varepsilon(m)}{\partial w_{ij}(m)} = \frac{\partial \varepsilon(m)}{\partial e_i(m)}\frac{\partial e_i(m)}{\partial y_i(m)}\frac{\partial y_i(m)}{\partial w_{ij}(m)} \tag{9}$$

Equation 10 is obtained from Eqs. (2), (1), and (5)

$$\frac{\partial \varepsilon(m)}{\partial e_i(m)} = e_i(m) \tag{10}$$

$$\frac{\partial e_i(m)}{\partial y_i(m)} = -1 \tag{11}$$

$$\frac{\partial y_i(m)}{\partial w_{ij}(m)} = f'\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right]\frac{\partial\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right]}{\partial w_{ij}(m)}$$

$$= f'\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right] y_i(m) \tag{12}$$

where

$$f'\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right] = \frac{\partial f\left[\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right]\right]}{\partial\left[\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right]\right]}$$

Substituting Eqs. (8), (9), and (10) in Eq. 9, the following expression arrives

$$\frac{\partial \varepsilon(n)}{\partial w_{ij}(m)} = -e_j(m)\, f'\left[\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\right] y_i(m) \tag{13}$$

Delta rule is used to provide the correction $\Delta w_{ij}(m)$, and it is expressed as

$$\Delta w_{ij}(m) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ij}(m)} \tag{14}$$

where η is a constant pre-determined parameter for the learning rate in the back-propagation algorithm.

5 Types of Deep Learning Networks

The deep learning network is classified into three classes depending upon the techniques and architectures used for a particular application like synthesis, classification, and recognition. They are classified into:

(i) Unsupervised deep learning network
(ii) Supervised deep learning network
(iii) Hybrid deep learning networks

Unsupervised deep learning network captures higher-order correlation data for synthesis purposes when there is no clear target class defined. In supervised learning of deep networks, discriminative power is provided for pattern classification by portraying the distributions of classes accustomed on the data which is visible. It is otherwise known as discriminative deep networks. A hybrid deep neural network exploits both discriminative and generative components. Moreover, a hybrid deep neural network model is structured by converging homogeneous convolution neural network (CNN) classifiers. The CNN classifiers are trained to yield an output as one for the predicted class and zero for all the other classes.

6 Deep Learning Architecture

In this deep learning architecture section, the commonly used deep learning approaches are discussed. Representation is a significant factor in deep learning. In the traditional method, the input features are extracted from raw data to be fed in machine learning algorithms. It relies on domain knowledge and the practitioner's expertise to determine the pattern. Traditional Software Engineering methodology like create, analyze, select, and evaluate are time-consuming and laborious. In contrast, the appropriate features are learned from the data directly without any human intervention and facilitate the discovery of dormant relationship among data that might be otherwise hidden or unknown.

In deep learning, the complex data representation is commonly expressed as compositions of simpler representations. Most of the deep learning algorithms are constructed based on the conceptual framework of artificial neural network (ANN), and it comprises interconnected nodes called as "neurons" which are organized in layers shown in Fig. 5. The neuron which does not exist in these two layers is called hidden units, and it stores the set of weights W.

Artificial neural network weights can be augmented by minimizing the loss function, for instance, negative log-likelihood, and it is denoted in Eq. 1:

$$E(\theta,D) = -\sum_{i=0}^{D}\left[\log P\left(Y = y_i|,x_i|,\theta\right)\right] + \lambda \quad \theta \quad p \tag{15}$$

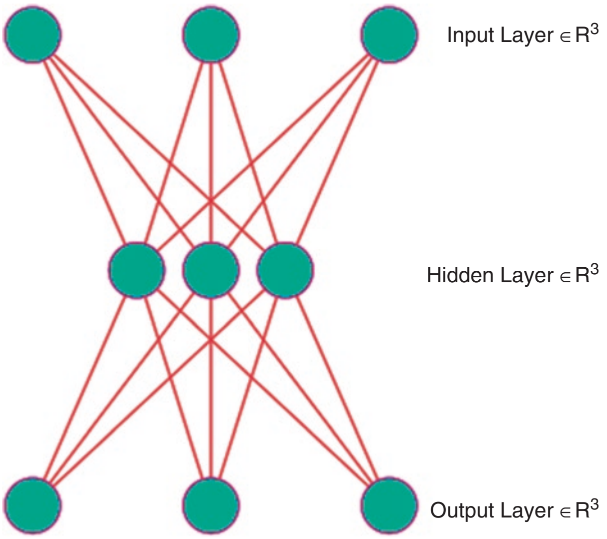


Fig. 5 Neural network with 1, 2, 1 input, hidden, and output layers

The first term minimizes the total log loss in the whole training dataset D.

The second term minimizes the p-norm of learned parameter θ_i, and it is controlled by λ a tunable parameter.

It is referred as regularization, and it prevents a model to be overfitting. Normally, the loss function can be optimized using a backpropagation mechanism, and it is meant for weight optimization that reduces the loss by traversing backward from the final layer in the network. Some of the deep learning open-source tools are Keras3, Theano2, TensorFlow1, Caffe6, DeepLearning4j8, CNTK7, PyTorch5, and Torch4. Some commonly used deep learning models discussed are based on optimization strategy and ANN's architecture. The deep learning algorithms are categorized into supervised and unsupervised techniques. The supervised deep learning architecture includes convolutional neural networks, multilayer perceptrons, and recurrent neural networks. The unsupervised deep learning architecture includes autoencoders and restricted Boltzmann machines (Fig. 6).

6.1 *Supervised Learning*

6.1.1 Multilayer Perceptron (MLP)

Multilayer perceptron holds many hidden layers; the neurons in the base layer *i* is completely connected to neurons in *i + 1* layer. Such type of network is restricted to have minimal hidden layers, and the data is allowed to transmit in one direction only. A weighted sum is computed for the outputs received from the hidden layer in each hidden unit. Equation 16 represents a nonlinear activation function σ of the computed sum. At this point, d refers to the number of units available in the previous layer, and x_j is referred as the output received from the previous layer j_{th} node. b_{ij} and w_{ij} are considered as bias and weight terms that are associated with each x_{ij}. Tanh or

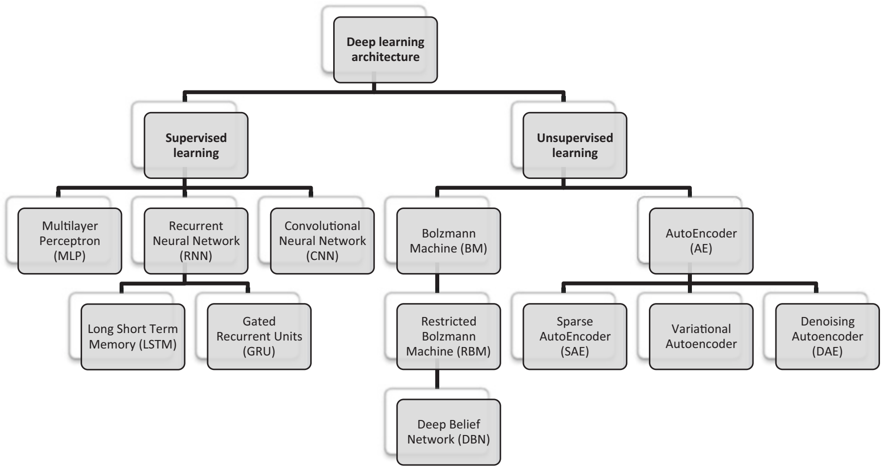

Deep learning architecture
Supervised learning
Unsupervised learning
Multilayer Perceptron (MLP)
Recurrent Neural Network (RNN)
Convolutional Neural Network (CNN)
Bolzmann Machine (BM)
AutoEncoder (AE)
Long Short Term Memory (LSTM)
Gated Recurrent Units (GRU)
Restricted Bolzmann Machine (RBM)
Sparse AutoEncoder (SAE)
Variational Autoencoder
Denoising Autoencoder (DAE)
Deep Belief Network (DBN)

Fig. 6 Deep learning architecture and output layers

sigmoid are taken as the nonlinear activation functions in the conventional network, and rectified linear units (ReLU) [8] are used in modern networks.

A multilayer perceptron comprises of multiple hidden layers where

$$hi = \sigma\left(\sum_{j=1}^{d} x_j w_{ij} + b_{ij}\right) \tag{16}$$

After optimizing hidden layer weights during training, a correlation among the input x and output y is learned. The availability of many hidden layers makes the input data representation in a high-level abstract view because of the hidden layer's nonlinear activations. It is one of the simplest models among other learning architectures which incorporate completely connected neurons in the final layer.

6.1.2 Recurrent Neural Network (RNN)

CNN is an appropriate choice if the input data has a neat spatial structure (e.g., collection of pixels in an image), and RNN is a logical choice if the input data is ordered sequentially (e.g., natural language or time series data). One-dimensional sequence is fed into a CNN; the output of the extracted features will be shallow [8], meaning only closed localized relationships among few neighbors are considered for feature representations. RNNs are capable of handling long-range temporal dependencies. In RNN, hidden state *ht* is updated based on the triggering of current input *xt* at a time *t* and the previously hidden state *ht-1*. Consequently, the final hidden state contains complete information from all of its elements after processing an entire sequence. RNN includes:

1. Long short-term memory (LSTM)
2. Gated recurrent units (GRU)

The symbolic representation of RNN is shown in Fig. 7, with its equivalent extended representation, for instance, three input units, three hidden units, and an output. The input time step is united with the present hidden state that depends on the previous hidden state.

RNN includes LSTM and GRU models, the most popular variants referred to as gated RNN. The conventional RNN consists of interconnected hidden units, whereas

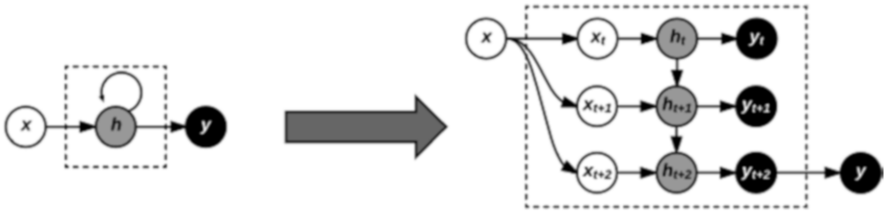


Fig. 7 RNN with extended representation

a gated RNN is substituted by a cell that holds an internal recurrence loop, and significantly the gates in this model control the information flow. The main advantage of gated RNN lies in modelling longer-term sequential dependencies.

6.1.3 Convolutional Neural Network (CNN)

CNN is a famous tool in recent years, particularly in image processing, and are stirred by the organization of the cat's visual cortex [5]. The local connectivity is imposed on the raw data on CNN. For example, more significant features are extracted by perceiving the image as a group of local pixel patches rather considering 50 x 50 image as individual 2500 unrelated pixels. A one-dimensional time series may also be viewed as a set of local signal segments. In particular, the equation for one-dimensional convolution is given as

$$C_{1\mathrm{d}} = \sum_{a=-\infty}^{\infty} \mathrm{x}(\mathrm{a}).\mathrm{w}(\mathrm{t}-\mathrm{a}) \tag{17}$$

where x refers to the input signal and w refers to the weight function or convolution filter.

The equation for two-dimensional convolution is given, where k is a kernel and X is a 2D grid:

$$C_{2d} = \sum_{m}\sum_{n} X(m,n)\,K(i-m,j-n) \tag{18}$$

The feature maps are extracted by calculating the weights of the input in a filter or a kernel. CNN encompasses sparse interactions considered as filters normally smaller than the input that results in less number of parameters. Parameter sharing is correspondingly encouraged in CNN because every filter is functional to the entire input. However, in CNN the same input is received from the previous layer which perfectly learns several lower level features. Subsampling is applied to aggregate the features which are extracted. The CNN architecture consists of two convolutional layers trailed by a pooling layer as depicted in Fig. 8. The application of CNNs is best in computer vision [6, 7].

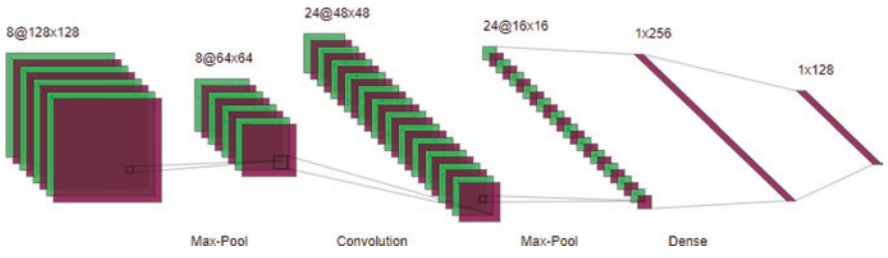


Fig. 8 ConvNet and output layers

6.2 Unsupervised Learning

6.2.1 Autoencoder (AE)

Autoencoder (AE) is the deep learning model that exemplifies the concept of unsupervised representation learning. Initially, it has pertained to supervised learning models once the labeled data was limited, but it is still remained to be useful for complete unsupervised learning such as phenotype discovery. In AE, the input is encoded into a lower-dimensional space z, and it is decoded further by reconstructing $\overline{x}$ of the corresponding input x. Hence, the encoding and decoding processes of an encoder are respectively given in equation with a single hidden layer. The encoding and decoding weights are represented as W and W0, and the reconstruction error is minimized. Z is a reliable encoded representation.

$$\mathrm{z} = \sigma\left(Wx + b\right) \tag{19}$$

$$\overline{x} = \sigma\left(W'z + b'\right) \tag{20}$$

As soon as an AE is well trained, then a single input is fed in the network and the innermost hidden layer activated to serve as input for the encoded representation (Fig. 9).

The input data is transformed into a structure where AE stores the utmost significant derived dimensions. It is similar to traditional dimensionality reduction techniques, namely, singular value decomposition (SVD) and principal component analysis (PCA). Deep autoencoder networks can be trained in a greedy manner, which is referred as the stacking process. Some of the autoencoder variants are:

1. Sparse autoencoder (SAE)
2. Variational autoencoder (VAE)
3. Denoising autoencoder (DAE)

6.2.2 Restricted Boltzmann Machine (RBM)

RBM is an unsupervised learning architecture that learns input data representation. It is almost similar to AE; instead, RBMs estimate the probability distribution of the available input data. Hence, it is perceived as a generative model where the data was generated in the underlying process. The canonical RBM is a model that consists of binary visible units $\vec{v}$, and hidden units $\vec{h}$ along with the energy function as shown in the equation:

$$E\left(v,h\right) = -b^T v - c^T v - Wv^T h \tag{21}$$

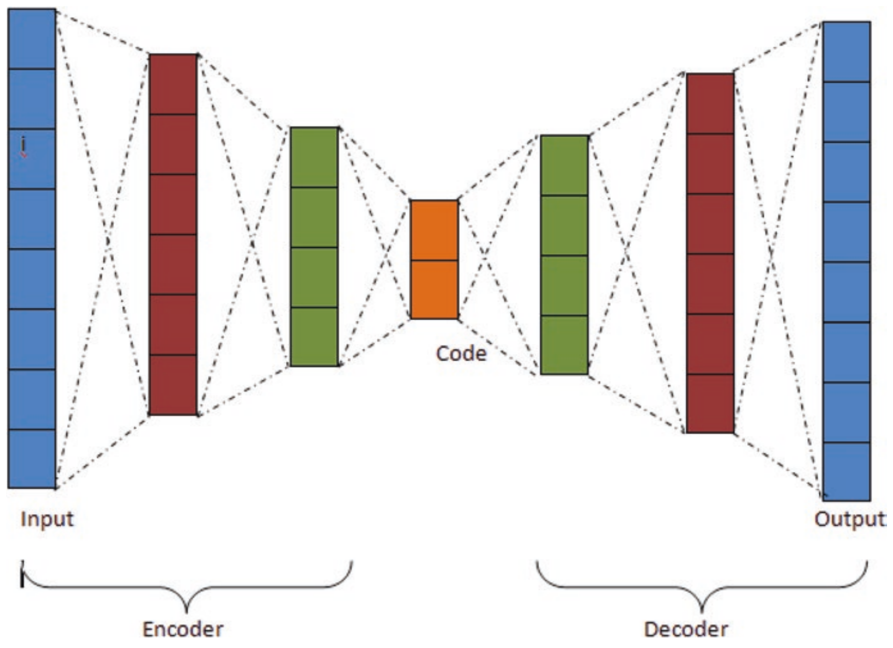


Fig. 9 Autoencoder example and output layers

In Boltzmann machine (BM), every unit is completely connected, while in restricted Boltzmann machine, there is no connection among the hidden units. Restricted Boltzmann machine is typically trained using a stochastic optimization like Gibbs sampling, and it yields the learned representation of the given input data that is viewed as the final form of h. Moreover, RBMs can be stacked hierarchically to construct a deep belief network (DBN) particularly for supervised learning.

7 Platforms for Deep Learning/Deep Learning Frameworks

Many software packages are available for researchers to ease the construction of deep learning architectures, but few years back non-deep learning professionals faced many difficulties to hadle the software packages. This circumstance lasted until Google introduced the DistBelief system in 2012. Following DistBelief, similar software packages like TensorFlow, Microsoft Cognitive Toolkit (previously CNTK), DeepLearning4j, Caffe, Torch, Keras, Neural Designer, H2o.ai, and Deep Learning Kit have extensively spurred the industry (Fig. 10).

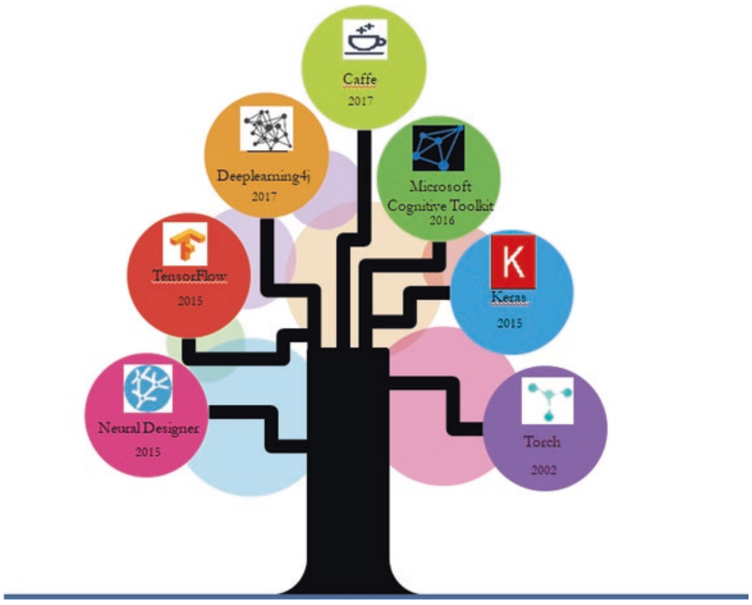


Fig. 10 Deep learning platform and output layers

7.1 TensorFlow

The concept of TensorFlow is highly associated with the mathematics involved in engineering and physics. Later TensorFlow has made its way to computer science which is associated with logic and discrete mathematics. Advanced machine learning concepts utilize the manipulation and calculus of tensors. TensorFlow is an open-source end-to-end machine learning library for production and research. It offers APIs for expert and beginner-level learners to develop applications for the cloud, mobile, web, and desktop. For beginner-level learners, TensorFlow recommends Keras API to develop and train the deep learning models. For advanced operations like forward passes, customizing layers, and training the loops with auto-differentiation, define-by-run interface API is recommended. Pre-made estimators are available to implement common machine learning algorithms. The architecture of TensorFlow is divided into four functioning parts, namely, data processor, model builder, training, and estimating the model. It accepts the inputs as tensors or multi-dimensional array, constructs operation flowchart which explains the multiple operations subjected to input, and finally comes out as output. Hence the name TensorFlow arises as the tensors flow through a list of operations and produce the desired output on the other side. TensorFlow is based on static graph computation, to visualize the constructed neural network with the help of TensorBoard. TensorFlow supports algorithms like classification, linear regression, deep learning wipe, deep learning classification, boosted tree classification, and boosted tree regression.

7.2 Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit is also a deep learning toolkit that considers neural networks as a sequence of computational procedure using a directed graph. The former version of this toolkit is the Computational Network Toolkit (CNTK). The latest version is CNTK v.2.0 Beta 1, available with new Python and C++ APIs using BrainScript as its own language. The Computational Network Toolkit libraries are developed using the C++ language. The Python APIs preserve abstractions for model definition, data reading, learning algorithms, and disbursed training. CNTK 2 is considered as the supplement of Python API with the feature of protocol buffer serialization developed by Google. CNTK 2 supports Fast R-CNN algorithm that supports the object-detection algorithm. Fast R-CNN algorithm is based on reusability concept that reuses computations from the convolution layers by adding a ROI pooling scheme. Microsoft Cognitive Toolkit is an open-source, multi-GPU machine, highly supportive for neural network training to classify and recognize images, text, and speech. Microsoft Cognitive Toolkit is the backbone for Skype live translation, Xbox, Bing, and Cortana. It supports a range of neural network types like convolutional neural network (CNN), feed-forward network (FFN), recurrent/long short-term memory (RNN/LSTM), sequence-to-sequence with attention, and batch normalization. Microsoft Cognitive Toolkit supports unsupervised learning, reinforcement learning, generative adversarial networks, and automatic hyper-parameter tuning. Parallelism can be achieved with the highest accuracy, even for the largest models in the GPU memory.

7.3 Caffe

Convolution Architecture For Feature Extraction (Caffe) is a deep learning framework developed by Berkeley AI Research with speed, expression, and modularity as its features. The speed of Caffe makes it suitable for industry development and research works. It can process nearly 60 M images per day with a single NVIDIA K40 GPU. The extensible code feature promotes active development. Expressive architecture encourages innovation and application. The usage of hard coding is minimized in model development.

7.4 DeepLearning4j

DeepLearning4j is a free and open-source deep learning library based on Java that provides complete solution for deep learning in various applications like deep predictive mining and knowledge discovery on CPUs and GPUs (graphics processing units). DeepLearning4j integrates the algorithms of artificial intelligence (AI) and techniques that are applicable for cyber forensics, business intelligence, robotic process automation (RBA), predictive analysis, network intrusion detection and prevention, face recognition, recommender systems, anomaly detection, regression, and many others. DeepLearning4j can import models from the advanced deep learning frameworks like Keras, Theano, Caffe, and TensorFlow. DeepLearning4j can initiate the interface for both Python and Java programming without any compatibility problem. The features of DeepLearning4j are based on microservice architecture. It provides scalability on Hadoop for big data and supports GPUs for scalability on Amazon Web Services (AWS) cloud. It is based on a distributed architecture with multi-threading; provides parallel computing and training and APIs for Java, Python, and Scala; and supports CPUs and GPUs, and massive amounts of data can be processed using clusters. The libraries and components associated with DeepLearning4j are ND4j, JavaCPP, DataVec, and RL4J. ND4j is the combined application of NumPy and Java virtual machine (JVM). ND4j is a library that provides rapid processing of matrix data, numerical computations, and performance-aware execution of multi-dimensional objects including linear algebra, signal processing, optimization, gradient descent, transformations, etc. JavaCPP is an interface and bridging tool for C++ and Java without ant third-party and intermediate applications. DataVec is a tool for ETL (extract, transform, and load) which facilitates the transformation of raw data to vector format with preprocessing to make it companionable for training in machine learning implementations. It supports binary, videos, images, text, CSV, etc. RL4J is the reinforcement learning for Java platforms with the integration of Deep Q-Learning, Asynchronous Actor-Critic Agents (A3C), etc.

7.5 Keras

Keras is an open-source neural network library developed by François Chollet, with features like fast, modular, and user friendly in Python platform that works on top of TensorFlow or Theano. Backend is a library within Keras to handle low-level computation as Keras is dedicated to advanced API wrapper. Backend performs the low-level computations like convolutions, tensor products with the aid of Theano or TensorFlow. It is capable of running on top of Theano, CNTK, or TensorFlow. Keras high-level API compiles the model which is designed with optimizer and loss functions and handles training process with fit function. Keras can support many platforms and devices, and it can be deployed in Web browser with .js support, Raspberry Pi, iOS with CoreML, Android with TensorFlow Android, and Cloud engine. Keras supports parallel data processing, and hence it handles huge volume of data and speeds up the training process.

7.6 Neural Designer

Neural Designer is a deep learning tool which is used to implement analytics algorithms and make it easy to handle. It is designed with a graphical user interface that defines the flow of work and gives accurate result. It is easy to handle as there is no programming or block diagrams involved. The user interface helps the user and instructs the procedure in a well-defined manner. Neural viewer is a visualization tool which displays the correct results in the form of exportable charts, tables, and pictures. Neural Designer has advanced algorithms that allow the users to construct incomparable predictive models. With the help of complicated data preprocessing methodology, the calculation of principal components and cleaning of outliers are made easy. In Neural Designer, the user can construct the most powerful predictive models with the help of various error and regularization methods. In addition to this, powerful strategies like Levenberg-Marquardt and quasi-Newton method are included to produce more precise computations. It also contains few strategies for testing the generalization capabilities of a predictive model. It holds higher processing speed and better memory management with CPU parallelization characteristics by means of GPU acceleration with CUDA and OpenMP.

7.7 Torch

Torch (Lua) is a Matlab-like environment for deep and not-so-deep machine learning solutions with flexible tensor implementation facility. Some of the features automatically compute gradients and hold multi-backend tensor for faster CPU/GPU computation, and rapid prototyping is possible as it supports high-level language.

8 Deep Learning Application

8.1 Speech Recognition

Speech recognition utilizes deep learning concepts and becomes the foremost application of deep learning by cautiously using its power. In 2010, deep learning makes its footprint in the speech recognition application. Gaussian mixture model (GMM) is the traditional speech recognition system which is based on hidden Markov models (HMMs). Here the speech signal is considered as short-time stationary signal or piecewise stationary signal, and hence Markov model is apt for this application. The limitation of this method is that it is inefficient for modelling nonlinear functions [20]. In contrast to HMMs, neural networks prove its efficiency in discriminative training. Neural networks provide better results for short-time signals; when it is continuous speech signals, the efficiency is questionable because it is not able to model temporal dependencies for continuous signals. In 2012, Microsoft announced a deep learning-based novel version of their Microsoft Audio Video Indexing Service (MAVIS). The result published by Microsoft clearly showed that deep learning-based application reduces word error rate (WER) when compared to Gaussian mixtures [21].

8.2 Deep Learning in HealthCare

Healthcare system is facing a new era by utilizing advanced technologies and providing right treatment for right patient at right time. Deep learning is the most powerful tool that allows a machine to learn from huge volume of data to make decisions. Researchers found that processing pharmaceutical information with multilayer neural networks produces accurate predictive decisions in various clinical applications. Deep learning architecture is based on hierarchical learning structure, and it has the capacity to integrate heterogeneous data to produce greater generalization. Many studies proved that deep learning paved its way toward the next-generation healthcare system that can accommodate billions of patient records to predict diseases, personalized prescriptions, clinical trials, and treatment recommendations. Using the temporal deep learning approach, Wang et al. have won the Parkinson's Progression Markers Initiative data challenge in identifying the subtypes of Parkinson's disease [9]. The traditional matrix- or vector-based approach is not considered as the best method as Parkinson's disease is vastly progressive and it is difficult to identify the disease progression patterns. Moreover, Wang et al. identified another three Parkinson's disease subtypes using LSTM RNN model, which demonstrates the dominance and potential of deep learning models in healthcare issues.

The deep learning architecture applicable for healthcare system mostly falls on recurrent neural networks (RNNs) [10], convolutional neural networks (CNNs) [11], autoencoders (AEs) [12], and restricted Boltzmann machines (RBMs) [13]. The major application of deep learning in healthcare system falls on image processing, especially to predict Alzheimer's disease using magnetic resonance imaging (MRI) scans [14, 15]. The risk of osteoarthritis can be detected using CNNs that represent low-field knee MRI hierarchically to automatically segment cartilage. Deep learning is used to segment multiple sclerosis lesions in multi-channel 3D MRI in order to predict malignant and benign breast nodules. Deep learning is being used to process electronic health record (EHR) data in the field of laboratory test, diagnosis, and medication and for free-text clinical notes. Area under the receiver operating characteristic curve and F-score methodology prove that deep learning accuracy is superior than traditional machine learning process [16]. DeepCare, a deep dynamic network, is a RNN with long short-term memory (LSTM) hidden units, pooling, and word embedding which identifies the current state of illness and predicts the future consequences [17]. Choi et al. [18] have developed Doctor AI model that predicts diagnoses and medications using RNNs with gated recurrent unit (GRU). Miotto et al. [19] used a three-layer stacked denoising autoencoder (SDA) and proposed a deep patient representation from the EHRs to predict risks based on random forest classifiers.

8.3 Deep Learning in Natural Language Processing

Natural language processing (NLP) is a computational methodology for automatic analysis of human language. Research activities in document text and language are increasing nowadays in the signal processing community. Deep learning contribution is in language modelling to give a probability to sequence of linguistic symbol or word. Natural language processing (NLP) started in the era of batch processing and punch card in that the analysis takes up to 7 minutes [22]. Deep learning algorithms and architectures are doing impressive advancements in the fields of pattern recognition. Collobert et al. [23] explained that a deep learning [24] framework can perform well in many natural language processing tasks such as semantic role labeling (SRL), named-entity recognition (NER), and POS tagging [25]. Following Collobert et al., various versions of advanced deep learning-based algorithms have evolved to resolve various difficult natural language processing tasks [26–28].

9 Conclusion

Deep learning is a branch of machine learning algorithm that is used to model data with the help of architectures consisting of multiple nonlinear transformations. This chapter explained the basis for deep learning, and its evolution is discussed with a roadmap. The architectures, learning methods, and its applications are elaborated. The platforms and their brands are discussed deeply with their applications. The general applications and the life-saving applications of deep learning in healthcare system are clearly explained. The future of deep learning is very bright, and the great thing about deep learning is that it is doing extremely well at dealing with vast amount of disparate data that are relevant in current era of smart sensors that collect huge amount of information.

References

1. Kuhn, Deanna, et al. Handbook of Child Psychology. Vol. 2, Cognition, Perception, and Language. Wiley, 1998
2. Restak, Richard M. and David Grubin. The Secret Life of the Brain. Joseph Henry Press, 2001
3. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
4. Mountcastle, V.B.: Modality and topographic properties of single neurons of cat's somatic sensory cortex. J. Neurophysiol. **20**(4), 408–434 (1957)
5. Hubel, D.H., Wiesel, T.N.: Receptive fields and functional architecture of monkey striate cortex. J. Physiol. **195**, 215–243 (1968)
6. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Adv Neural Inf Process Syst, 1097–1105 (2012)

7. Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1–9
8. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
9. The Michael J. Fox Foundation for Parkinson's Research: subtyping Parkinson's disease with deep learning models. https://www.michaeljfox.org/foundation/grant-detail.php
10. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Comput. **1**, 270–280 (1989)
11. Lecun, Y., Bottou, L., Bengio, Y., et al.: Gradient-based learning applied to document recognition. Proc. IEEE. **86**, 2278–2324 (1998)
12. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science. **313**, 504–507 (2006)
13. Smolensky P. Information processing in dynamical systems: Foundations of harmony theory (No. CU-CS-321-86). Colorado University at Boulder Dept of Computer Science 1986
14. Liu S, Liu S, Cai W, et al. Early diagnosis of Alzheimer's disease with deep learning. In: International Symposium on Biomedical Imaging, Beijing, China 2014, 1015–1018
15. Brosch, T., Tam, R.: Manifold learning of brain MRIs by deep learning. Med Image Comput Comput Assist Interv. **16**, 633–640 (2013)
16. Manning, C.D., Raghavan, P., Schutze, H.: Introduction to Information Retrieval, vol. 3. Cambridge university press, Cambridge (2008)
17. Pham T, Tran T, Phung D, et al. Deep Care: a deep dynamic memory model for predictive medicine. arXiv 2016. https://arxiv.org/abs/1602.00357
18. Choi E, Bahadori MT, Schuetz A, et al. Doctor AI: predicting clinical events via recurrent neural networks. arXiv 2015. http://arxiv.org/abs/1511.05942v11
19. Miotto, R., Li, L., Kidd, B.A., et al.: Deep patient: an unsupervised representation to predict the future of patients from the electronic health records. Sci. Rep. **6**, 26094 (2016)
20. Singh, H., Bathla, A.K.: A survey on speech recognition. Int. J. Adv. Res. Comput. Eng. Technol. **2**(6), 2186–2189 (2013)
21. Xie, Y., Le, L., Zhou, Y., Raghavan, V.V.: Deep learning for natural language processing. In: Handbook of statistics. Elsevier, Amsterdam, The Netherlands (2018)
22. Cambria, E., White, B.: Jumping NLP curves: a review of natural language processing research. IEEE Comput. Intell. Mag. **9**(2), 48–57 (2014)
23. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. J Machine Learn Res. **12**(Aug), 2493–2537 (2011)
24. Prakash, K. B., Ruwali, A., Kanagachidambaresan, G. R.: Introduction, in to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
25. JHA, A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor Flow Basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
26. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57007-4_3
27. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Regression, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57007-4_4
28. Vadla, P.K., Ruwali, A., Lakshmi, M.V.P., Kanagachidambaresan, G.R.: Neural network, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57007-4_5

Ms. Indrakumari is working as an Assistant Professor in School of Computing Science and Engineering, Galgotias University, NCR Delhi, India. She has completed M.Tech in Computer and Information Technology from Manonmaniam Sundaranar University, Tirunelveli. Her main thrust areas are big data, Internet of Things, data mining, and data warehousing and its visualization tools like Tableau, QlikView.

Dr. T. Poongodi is working as an Associate Professor in School of Computing Science and Engineering, Galgotias University, NCR Delhi, India. She has completed Ph.D in Information Technology (Information and Communication Engineering) from Anna University, Tamil Nadu, India. Her main thrust research areas are big data, Internet of Things, ad hoc networks, network security, and cloud computing. She is a pioneer researcher in the areas of big data, wireless network, and Internet of Things and has published more than 25 papers in various international journals. She has presented a paper in national/international conferences; published book chapters in CRC Press, IGI Global, and Springer; and edited books.

Ms. Kiran Singh is presently working as an Assistant Professor in the Department of Computer Science and Engineering at Galgotias University. She received her MCA degree from Maharshi Dayanand University in 2008 and M.Tech in Computer Science and Engineering from Rajiv Gandhi Proudyogiki Vishwavidyalaya in 2015, Bhopal. She has overall experience of 11 years. Her research interests include image processing, big data, and IOT. She has published papers in international journal and conference.

Deep Learning Applications with Python

Nahil Ahmed Hassan, Abhigith Neil Abraham, and Ajeesh Ramanujan

1 Introduction

One of the most desirable features of a programming language used for working with the deployment of deep learning models would be the ability for quick prototyping with minimal effort. This, along with other myriads of benefits, has made Python incredibly popular among the deep learning community. The presence of comprehensive frameworks for data preprocessing, visualization, and model development within the Python environment has made deep learning workflow simpler and also to be executed in a few lines of code.

Python is renowned for its coherent, readable code and is practically unmatched in terms of ease of use and clarity, particularly for beginners in the field. The simplicity in programming allows the developers to implement the complex algorithms and networks in deep learning without the hassle of knowing the inner working of the language. This feature also facilitates faster development and comprehensive testing for developing models, which is of prime importance in this domain.

Extensibility is another feature that makes Python suitable for deep learning applications. The abundance in the collection of modules and libraries that facilitate the various stages in the workflow of a deep learning problem makes the language more productive for developers as development time, and coding, decreases substantially. The excellent compatibility of popular deep learning frameworks like

N. A. Hassan (✉) · A. Ramanujan
Department of Computer Science, College of Engineering Trivandrum, Thiruvananthapuram, India
e-mail: ajeesh@cet.ac.in

A. N. Abraham
Department of Electrical Engineering, College of Engineering Trivandrum, Thiruvananthapuram, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_2

TensorFlow and PyTorch with Python along with the availability of libraries like NumPy and SciPy for scientific computation, Matplotlib for visualizations, and Scikit-Learn for data analysis makes the language an ideal playground for learning and solving deep learning problems.

Being an interpreted language, Python is slower compared to other compiled languages like C++ and Java. This limitation concerning the execution time can be constraining in circumstances where high performance is a necessity. However, efficient data structures and reduced development time can, to an extent, make up for the deficiency in execution time.

The following sections in this chapter will introduce different applications that deep learning has had a significant influence on the recent past. Each section comprises a brief introduction of the problem, the approaches used to solve them in the past, and how deep learning methods have been able to obtain better results. Following this, each section will have a full practical Python implementation of the problem encompassing the entire workflow for dealing with the challenge.

2 Deep Learning for Face Recognition

2.1 Brief Introduction

Facial recognition is one of the most prominent biometric techniques used for identity authentication and verification. It is the identification of an individual based on the photograph of their face. Facial recognition techniques are extensively used in areas like public security, social media, and other commercial domains in daily life. A facial recognition system will automatically identify faces present in a still image or even a video. The vast increase in the availability of cheap and powerful embedded devices has generated a tremendous rise in face recognition applications and research [18, p. 1].

Right from the 1990s, facial recognition generated immense scientific curiosity among the computer vision researchers. One of the first methods in face recognition to attain popularity was the Eigenface approach [38]. But this approach was unable to address the uncontrollable variations in faces from the model's assumptions. This disadvantage led to the emergence of local feature-based models like Gabor [28] and LBP [2], which overcame the weaknesses of the previous models through local filtering. However, all these works struggled with sophisticated variations in facial appearance.

Deep learning models, especially convolutional neural networks (CNNs), gained immense popularity after AlexNet [26] won the ImageNet competition. The ability of CNNs to capture and learn various representation levels at different abstractions provides strong invariance to different facial appearance variations. In 2014, DeepFace [37] was able to achieve near human-level performance on the LFW [16] benchmark dataset with a 9-layer CNN trained using a dataset containing 4 million facial images. Following this, several deep learning-based research on facial recognition arose, which eventually led to a tremendous improvement in accuracy over a short period. VGGFace [33], VGGFace2 [5], and FaceNet [35] are some of the other prominent works in facial recognition which have been able to generate state-of-the-art results.

2.2 Datasets

The presence of a large-scale database is a necessary condition for the effective working of a facial recognition system. With saturation in the performance over simple databases like LFW [16], researchers started developing large-scale complex databases for obtaining better results. Most of the early state-of-the-art works like DeepFace [37] and FaceNet [35] were trained using large-scale private databases. This situation led to researchers being unable to reproduce the results these works attained or to compare their models due to lack of public datasets.

CASIA-Webface [42] was the first exhaustively used public dataset for facial recognition consisting of 0.5 M images of 10,000 celebrities. More and more public datasets with a large number of images for training deep models made an appearance over time. Datasets like MS-Celeb-1 M [14], VGGFace2 [5] and MegaFace [23], consist of over 1 M images and are extensively used for facial recognition research.

2.3 Practical Example

In this section, a ResNet50 [15, p.775] model trained using the VGGFace2 dataset is used to perform facial recognition. The authors of VGGFace2 have open-sourced their models and have released implementations for popular frameworks like Caffe and PyTorch, but not for TensorFlow or Keras. The **keras-vggface** [29] is an open-source python-based library that has implemented VGGFace and VGGFace2 pre-trained models in Keras. It also provides various utility functions for preprocessing images and decoding predictions.

The library can be installed using the pip command.

```
>>> pip install keras_vggface
```

MTCNN or “multi-task cascaded convolutional neural network” [19] is one of the popularly used models for face detection within images, which has achieved excellent results. The **mtcnn** [6] library in Python loads the Keras implementation of the MTCNN model into the Python program.

This can be installed using the pip command as well.

```
>>> pip install mtcnn
```

On the successful installation of these libraries, the necessary libraries are imported.

```
import keras_vggface
import mtcnn
import matplotlib.pyplot as plt
from PIL import Image
from numpy import asarray
from mtcnn.mtcnn import MTCNN
from keras import layers, Model
```

Now a function for detecting faces from an image should be implemented. The function will take two parameters, namely, **img** and **req_size**, as arguments. The img argument requires the image array to be used for face extraction, and the **req_size** parameter specifies the size to which the extracted face image must be resized. For using the mtcnn library, we must instantiate the network by calling the **MTCNN()** constructor. The mtcnn library provides a utility function called detect_faces(), which allows direct detection of faces from an image. This function returns a list of JSON objects where the keys of these objects are "*box*," "*confidence*," and "*keypoints*." The values of the "box" JSON object will provide us with the coordinates of the bounding box in the form [x, y, width, height]. These coordinates are used to extract the face for detection from the image.

```
#function to extract face from an image
def face_detect(img,req_size):
   #Initialising the MTCNN model with default weights
   model = MTCNN()
   #Detecting the faces in the image
   result = model.detect_faces(img)
   #Extracting the coordinates of the bound box
   x, y, width, height = result[0]['box']
   #Extracting the face from the image
   face = img[ y : y + height, x : x + width]
   #Resizing the result to our required size
   pixels = Image.fromarray(face)
   image = pixels.resize(req_size)
   image = asarray(image)
   return image
```

Now, we choose an image of actress Emma Atkins. Let the input image be named "emma-atkins.jpg" as shown in Fig. 1. The image is loaded into a NumPy array using the **plt.imread()** function. This array along with the required size of (224,224) is passed on as arguments to the function call for detection of the face from the image.

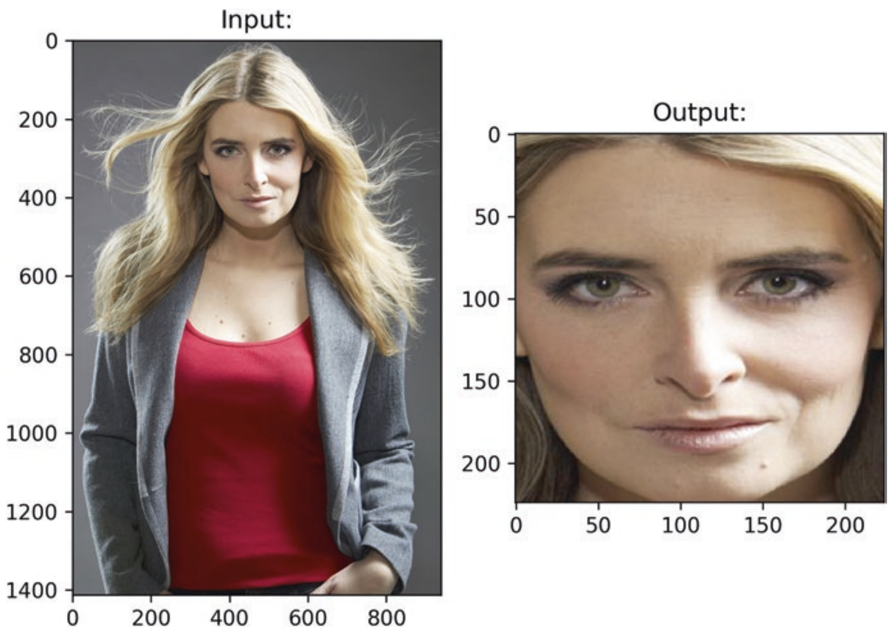


Fig. 1 Input image along with extracted output image

```
# loading the image
img = plt.imread('emma-atkins.jpg')
# function call with image and required size passed as argument
face = face_detect(img,(224,224))
# Plotting the input image along with extracted face
f = plt.figure(figsize=(7,7))
f.add_subplot(1,2,1)
plt.title("Input:")
plt.imshow(img)
f.add_subplot(1,2,2)
plt.imshow(face)
plt.title("Output:")
# show the plot
plt.show()
```

Now that the face has been extracted from the image loaded, facial recognition is carried out on the output photograph in Fig. 1. For this, the pretrained ResNet50 model is loaded using the **keras-vggface** library. The **VGGFace()** constructor from the library is called with the model parameter set to "*resnet50*" to instantiate the VGGFace model.

```
# importing the VGGFace() constructor
from keras_vggface.vggface import VGGFace
# load the pretrained resnet50 model
model = VGGFace(model='resnet50')
# previewing the input and output shape of the loaded model
print('VGGFace Input Shape:', model.inputs[0].shape)
print('VGGFace Output Shape:', model.outputs[0].shape)
```

```
>>>      VGGFace Input Shape: (None, 224, 224, 3)
         VGGFace Output Shape: (None, 8631)
```

```
from keras_vggface.utils import preprocess_input
from keras_vggface.utils import decode_predictions
from numpy import expand_dims

face = face.astype('float32')
face_array = expand_dims(face, axis=0)
face_array = preprocess_input(face_array, version=2)
# carrying out the prediction of the image
yhat = model.predict(face_array)
# converting predictions into identities
preds = decode_predictions(yhat)
# printing out the top five predictions
for prediction in preds[0]:
 print('{0}:
{1:3.3f}%'.format(prediction[0].replace("b'","").replace("'",""),
 prediction[1]*100))
```

```
>>>     Emma_Atkins: 98.658%
        Ir\xc3\xa1n_Castillo: 0.171%
        Gr\xc3\xa1inne_Seoige: 0.109%
        Cobie_Smulders: 0.050%
        Delphine_Batho: 0.036%
```

The output of the above code snippet specifies that there are 8631 neurons in the final dense layer of the VGGModel. This observation is in accordance with the fact that the VGGFace2 models were trained using the 8631 output classes in the MS-Celeb-1 M dataset.

Following this, two utility functions, **preprocess_input** and **decode_predictions**, have to be imported from the keras-vggface library. The **preprocess_input** scales the pixel values of the image in the same way the data was scaled during the training of the VGGFace model. After the extracted face has been preprocessed using the preprocess_input function, the identity of the image is predicted using the **model.predict()** function. This returns a NumPy array that contains the output probability values of each of the 8631 output neurons in the final dense layer. The **decode_predictions** function maps this NumPy array to their corresponding celebrity name label and also fetches the top five celebrities with the highest probabilities.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from imgaug import augmenters as iaa
```

It can be seen that the pretrained ResNet50 model was able to correctly identify the face in the image as that of Emma Atkins with a probability of **98.658%**.

To understand how the features are extracted from an input image, it is intuitive to visualize the convolutional neural network's intermediate outputs. These visualizations are useful in understanding how successive convolutional layers transform their input and what each filter does to the input image that the layer receives.

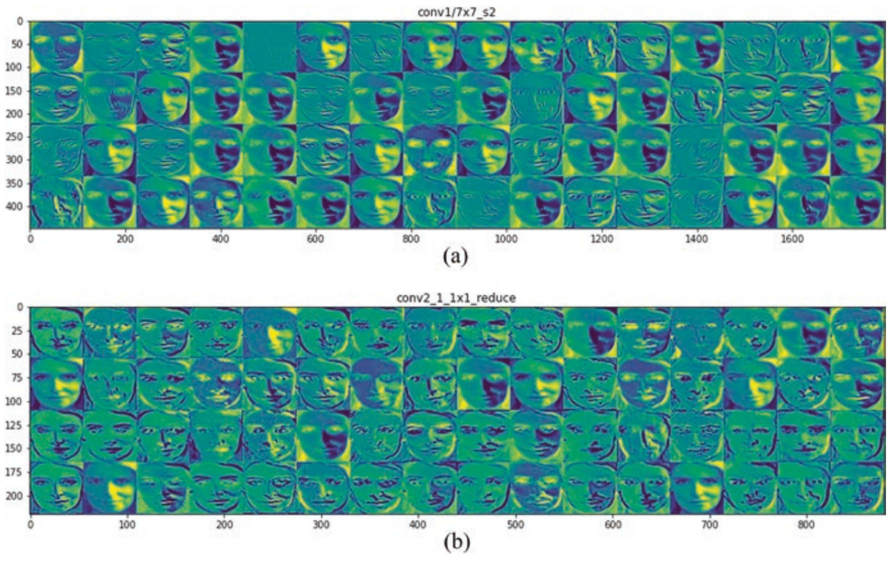


Fig. 2 (**a**, **b**) Visualization of all the channels of the first two convolutional layers of the network

Figure 2 shows the visualizations of the intermediate outputs of the first two convolutional layers of the pretrained VGGFace model when the input image is fed forward into it. It can be inferred that the features extracted by a particular layer become more and more abstract to the depth of the layer, i.e., the first convolution layer retains almost all the information present in the input image, but careful analysis of the intermediate outputs of the last layers reveals them to be much less capable of being interpreted and carrying less information pertinent to the input image fed into the network.

Figure 3 depicts the output from the layer "conv5_3_3x3," which consists of 512 filters, that is placed at the tail end of the ResNet50 network used in this example. This intermediate output carries very little information from the input image we fed forward but has increasingly more information related to the class or target label of the picture. The networks continuously filter out the unnecessary features and focus more on the useful information within the image, which helps the network in classifying.

3 Deep Learning for Fingerprint Recognition

3.1 Brief Introduction

Fingerprint recognition refers to the identification of an individual based on the comparison of two fingerprints. Fingerprint-based identification is considered one of the most reliable biometric techniques since it is virtually impossible to forge a

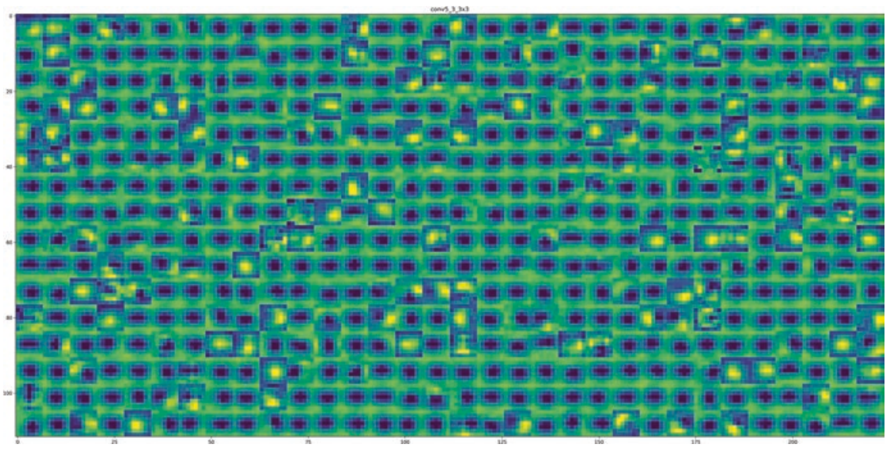


Fig. 3 Visualization of all the channels of the layer "conv5_3_3x3"

signature of a particular identity. The technique is widely used in the field of forensics, airport security, etc. The automated fingerprint identification system is a system that utilizes digital image processing capabilities to obtain, store, and analyze fingerprint data. Such a system also allows the matching of one fingerprint with another for recognition and identification purposes. With the advent of novel methods and techniques to match fingerprints without any kind of human assistance, fingerprint recognition has had significant improvement over its former iterations.

Many of the solutions and research upon the fingerprint recognition problem revolve around the regular minutiae-based matching. There have also been works based upon the construction of hand-crafted features followed by the pairing of the fingerprint. These methods require extensive data preprocessing and the usage of several hand-crafted features, which may not be optimal while dealing with datasets containing a large number of images and output classes.

Deep learning methods, in the recent past, have had great success in the tasks of image recognition and classification. Especially convolutional neural networks (CNNs) have been able to extract features that far outperform the models based upon conventional hand-made features. There have been numerous works where CNNs have been applied to the task of fingerprint recognition which has been able to produce excellent results. Shrein [45] used a network comprising five convolution layers to produce 95.9% accuracy on the NIST-DB4 dataset. Other similar works upon fingerprint recognition include [9, 34], etc.

3.2 Datasets

Fingerprint recognition datasets consist of fingerprint data obtained from live-scan sensors and digitized fingerprint impressions from documents. The FVC databases [11] are databases that were created for each FVC competition that was organized in the years 2000, 2002, 2004, and 2006. These datasets were obtained using different sensors like low-cost optical sensors, low-cost capacitive sensors, etc. Another popular set of public datasets used for fingerprint recognition are the ones released by the National Institute of Standards and Technology (NIST). NIST has released several special biometric databases like NIST-DB4, NIST-DB9, and NIST-DB14, where each one is focused upon a different challenge pertinent to recognition. There have also been other prominent public datasets that are able to incorporate the traits acquired by live-scan sensors like the MCYT Bimodal Database [32], BIOMET Multimodal Database [12], and several others.

3.3 Practical Example

In this section, a Siamese convolutional neural network [24] is used to perform fingerprint recognition. The network comprises identical CNNs that share the same weights. A pair of images of the same class or different classes is provided as input to the training of the network. The similarity criterion of the input image pairs is learned by the network with the aid of a contrastive loss function. It would be much less computationally expensive to train a Siamese CNN over a conventional CNN from scratch for a dataset of average size.

The training of the model is done using the Sokoto Coventry Fingerprint Dataset [36]. The dataset consists of 6000 fingerprint images from 600 African subjects. The dataset also consists of synthetically altered versions of these fingerprints. A sample fingerprint image of a random identity along with its altered versions is shown in Fig. 4. The dataset is freely available for research purposes at https://www.kaggle.com/ruizgara/socofing. The altered images in the dataset have been classified into three based on the difficulty in the parameter settings used for the alteration, namely, easy, medium, and hard.

To begin with, all the necessary libraries are imported. The imgaug is a python library that supports a wide range of augmentation techniques on images. The train_test_split function in the sklearn library is used to split an array into train and test subsets randomly.

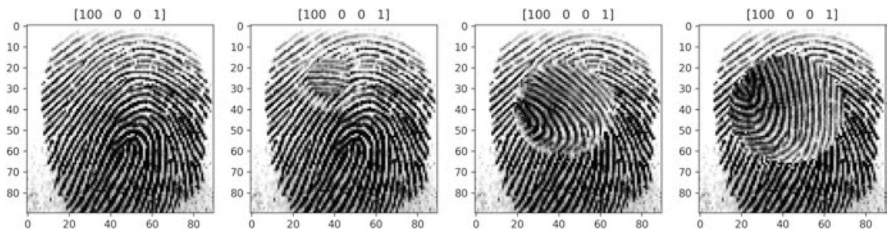


Fig. 4 Sample fingerprint along with its altered versions and their corresponding labels

```
x_real = np.load('dataset/x_real.npz')['data']
y_real = np.load('dataset/y_real.npy')
x_easy = np.load('dataset/x_easy.npz')['data']
y_easy = np.load('dataset/y_easy.npy')
x_medium = np.load('dataset/x_medium.npz')['data']
y_medium = np.load('dataset/y_medium.npy')
x_hard = np.load('dataset/x_hard.npz')['data']
y_hard = np.load('dataset/y_hard.npy')

x_data = np.concatenate([x_easy, x_medium, x_hard], axis=0)
y_data = np.concatenate([y_easy, y_medium, y_hard], axis=0)

x_train, x_val, label_train, label_val = train_test_split(x_data,
y_data, test_size=0.1)

print(x_data.shape, y_data.shape)
print(x_train.shape, label_train.shape)
print(x_val.shape, label_val.shape)
print(x_real.shape, y_real.shape)

>>>    (6000, 96, 96, 1) (6000, 4)
       (49270, 96, 96, 1) (49270, 4)
       (44343, 96, 96, 1) (44343, 4)
       (4927, 96, 96, 1) (4927, 4)
```

Once the libraries are imported, the dataset has to be loaded onto the program. The synthetically altered images and the real ones with their corresponding labels are loaded using the **np.load()** function. The labels are loaded as a set of NumPy arrays of the form [subject_id, gender, left/right, finger]. These label attributes are extracted from the filename of each image. All the synthetically altered pictures and their corresponding labels are grouped into NumPy arrays **x_data** and **y_data**, respectively. These NumPy arrays are provided as input into the **train_test_split()** function to split these fingerprint images and their corresponding labels into training and validation sets.

```
# creating a mapping between index and label
match_dict = {}
for i, y in enumerate(y_real):
   key = y.astype(str)
   key = ''.join(key).zfill(6)
   match_dict[key] = i
```

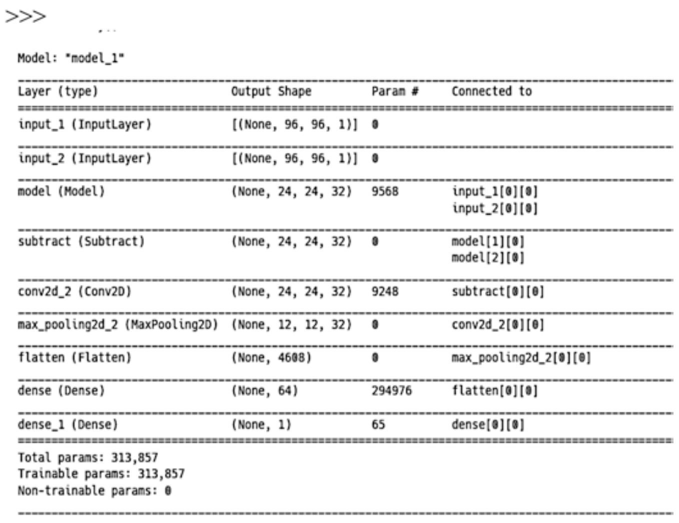
In order to create a mapping between the index of the images contained in the **x_real** dataset and their corresponding labels, a dictionary is created. This will come in handy while preparing batches of data for training.

Since Siamese CNN is being used, batches of images need to be generated for training. To implement this, a custom data generator class of type **tf.keras.utils. Sequence** is defined to specify how each batch of data will be generated from the dataset for training. The batch of images to be fed into the Siamese network must be created in such a way that a pair of images is chosen at random from the batch and its corresponding target variable Y_i is given an appropriate value based on whether the pair of fingerprints is of the same identity or not. The **__getitem__** function of the class allows us to implement this. This function also enables us to implement the various augmentation techniques provided by the **imgaug** library on the batch generated allowing the model to be less prone to overfitting. The **on_epoch_end** function is used to shuffle the batch of images at the end of each epoch which will eventually make the model much more robust. For further explanation of the functions used, refer to the **tf.keras.utils.Sequence** official documentation.

```
class DataGenerator(tf.keras.utils.Sequence):
  # constructor function
 def __init__(self, x, y, x_real, match_dict, batch_size=32, aug=True):

   self.x = x
   self.y = y
   self.x_real = x_real
   self.match_dict = match_dict
   self.batch_size = batch_size
   self.aug = aug

   self.on_epoch_end()

 # specifies the no of batches in the sequence
 def __len__(self):
   return int(np.floor(len(self.x) / self.batch_size))

 # generation of a batch of dataset for training
 def __getitem__(self,index):
   inp_img1 = self.x[ index*self.batch_size : (index+1)*self.batch_size ]
   img1_label = self.y[ index*self.batch_size : (index+1)*self.batch_size ]

   inp_img2 = np.empty((self.batch_size, 96, 96, 1), dtype=np.float32)
   batch_label = np.zeros((self.batch_size, 1), dtype=np.float32)

   # augmentation using the imgaug library
   if self.aug:
     seq = iaa.Sequential([
         iaa.GaussianBlur(sigma=(0, 0.5)),
         iaa.Affine(
             scale={"x": (0.9, 1.1), "y": (0.9, 1.1)},
             translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)},
             rotate=(-30, 30),
             order=[0, 1],
             cval=255
           )
     ], random_order=True)

     x1_batch = seq.augment_images(inp_img1)

   # preparing pairs of images by matching and unmatching
   for i, l in enumerate(img1_label):
     key = l.astype(str)
     key = ''.join(key).zfill(6)

     #randomly generating matched or unmatched pairs
     if random.random() > 0.5:
       # get matched image with same key
       inp_img2[i] = self.x_real[self.match_dict[key]]
       batch_label[i] = 1.
     else:
       # get unmatched image with different key
       while True:
         unmatch_key, index = random.choice(list(self.match_dict.items()))
         if unmatch_key != key:
           break

       inp_img2[i] = self.x_real[index]
       batch_label[i] = 0.

   #return the batch generated
   return [inp_img1.astype(np.float32) / 255., inp_img2.astype(np.float32) / 255.], batch_label

 # to shuffle the order of the examples at the end of each epoch
 def on_epoch_end(self):
   if self.aug == True:
     self.x, self.y = shuffle(self.x, self.y)
```

The Siamese network defined has two identical CNNs with the same weights. Each CNN is composed of two sets of a block of a convolution layer with 32 filters of dimension (3,3) followed by a max pool layer with a filter of dimension (2,2) to reduce the dimensionality of the image. The input to each of these networks is fingerprint images of size (96*96). Once a pair of images is passed through these subnetworks, the corresponding image feature tensors extracted are subtracted. They are then passed on to another block of a convolution layer, followed by a max pool layer with the same previous configurations. The Flatten layer converts the (n * n) dimensional image to a vector of dimensions (n^2 * 1). The following fully connected layer takes this (n^2 * 1) dimensional vector as input with a ReLU activation. Finally, the Dense layer is connected to the final layer, which consists of a single neuron and a softmax activation function to produce the final output. The model produces an output of 1 if the pair of fingerprints belongs to the same identity else produces 0. The architecture of the proposed model is displayed in the output using the **model.summary()** function.

```
x1 = layers.Input(shape=(96, 96, 1))
x2 = layers.Input(shape=(96, 96, 1))

# share weights both inputs
inputs = layers.Input(shape=(96, 96, 1))

feature = layers.Conv2D(32, kernel_size=3, padding='same', activation='relu')(inputs)
feature = layers.MaxPooling2D(pool_size=2)(feature)

feature = layers.Conv2D(32, kernel_size=3, padding='same', activation='relu')(feature)
feature = layers.MaxPooling2D(pool_size=2)(feature)

feature_model = Model(inputs=inputs, outputs=feature)


# 2 feature models that sharing weights
x1_net = feature_model(x1)
x2_net = feature_model(x2)

# subtract features
net = layers.Subtract()([x1_net, x2_net])

net = layers.Conv2D(32, kernel_size=3, padding='same', activation='relu')(net)
net = layers.MaxPooling2D(pool_size=2)(net)

net = layers.Flatten()(net)

net = layers.Dense(64, activation='relu')(net)

net = layers.Dense(1, activation='sigmoid')(net)

model = Model(inputs=[x1, x2], outputs=net)

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])

model.summary()
```



```
>>>
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 96, 96, 1)]  0
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 96, 96, 1)]  0
__________________________________________________________________________________________________
model (Model)                   (None, 24, 24, 32)   9568        input_1[0][0]
                                                                 input_2[0][0]
__________________________________________________________________________________________________
subtract (Subtract)             (None, 24, 24, 32)   0           model[1][0]
                                                                 model[2][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 24, 24, 32)   9248        subtract[0][0]
__________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 12, 12, 32)   0           conv2d_2[0][0]
__________________________________________________________________________________________________
flatten (Flatten)               (None, 4608)         0           max_pooling2d_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 64)           294976      flatten[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1)            65          dense[0][0]
==================================================================================================
Total params: 313,857
Trainable params: 313,857
Non-trainable params: 0
__________________________________________________________________________________________________
```

Now that both the model and the data have been set up as required, the training of the network can be done using the **model.fit()** function with the training and validation objects of the custom data generator class passed as parameters. The number of epochs is set to 10.

```
train_gen = DataGenerator(x_train, label_train, x_real, match_dict, aug=True)
val_gen = DataGenerator(x_val, label_val, x_real, match_dict, aug=False)
history = model.fit(x = train_gen, epochs=10, validation_data = val_gen)
```

At the end of ten epochs, the Siamese CNN achieves a training accuracy of 97.87% and a validation accuracy of 98.47%. Now, the results are plotted using the history callback provided by Keras. The callback keeps track of various metrics like the loss and accuracy for both training and validation datasets. The history object is returned when the **model.fit()** function is called upon. The resultant loss and accuracy curves are shown in Fig. 5 (Fig. 6).

```
# plotting the loss and accuracy curves
f = plt.figure(figsize=(12,5))
f.add_subplot(1,2,1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
f.add_subplot(1,2,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

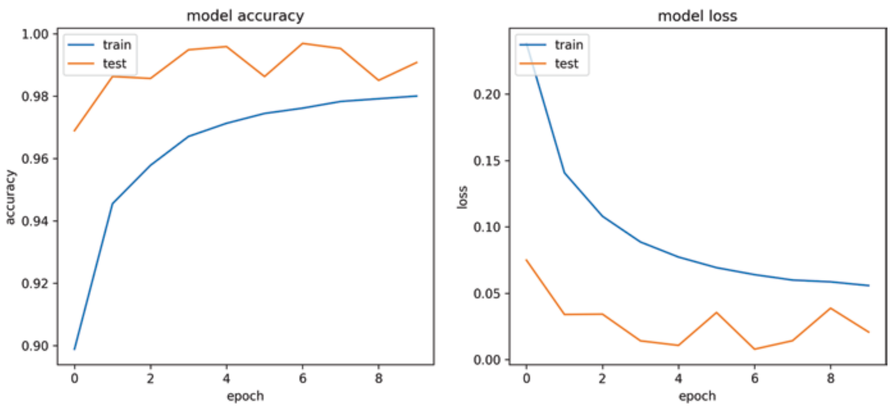


Fig. 5 Training and validation accuracy and loss curves

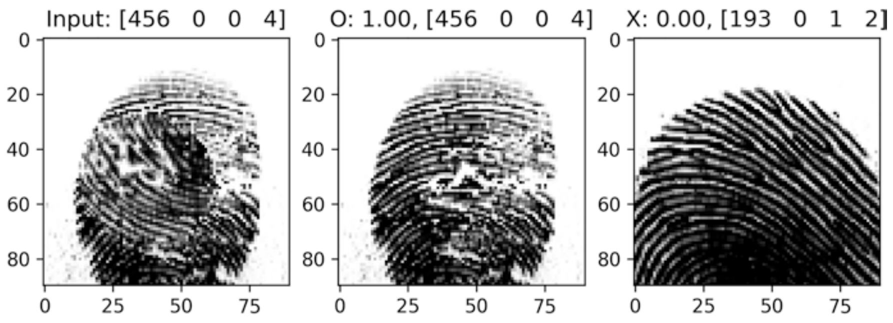


Fig. 6 Plot of the random image selected along with both its matched and unmatched pairs. The model output is also shown alongside each fingerprint

In order to test the Siamese CNN trained in this example, a random fingerprint will be chosen from the validation set, and a couple of pairs of images will be created from the chosen image with one pair belonging to the same identity while the other with another random identity. Both these pairs are fed forward into the network, and the corresponding results are plotted. The model produced an output of 1 for the pair of fingerprints where the identities matched and 0 for the one which didn't.

```
# choosing a random image and label for testing and reshaping
random_index = random.randint(0, len(x_val))
test_img = x_val[random_index]
test_label = label_val[random_index]
test_img = test_img.reshape((1, 96, 96, 1)).astype(np.float32) / 255.

# generating a matched pair with the test image
matched = test_label.astype(str)
matched = ''.join(matched).zfill(6)
matched_x = x_real[match_dict[matched]].reshape((1, 96, 96, 1)).astype(np.float32) / 255.
matched_y = y_real[match_dict[matched]]

# feed-forwarding the matched pair
pred_matched = model.predict([test_img, matched_x])

# generating an unmatched pair with the test-image
while True:
 unmatched, index = random.choice(list(match_dict.items()))
 if unmatched != matched:
   break
unmatched_x = x_real[index].reshape((1, 96, 96, 1)).astype(np.float32) / 255.
unmatched_y = y_real[index]

# feed-forwarding the unmatched pair
pred_unmatched = model.predict([test_img, unmatched_x])

# plotting the outputs produced by the model
plt.figure(figsize=(8, 4))
plt.subplot(1, 3, 1)
plt.title('Input: %s' %test_label)
plt.imshow(test_img.squeeze(), cmap='gray')
plt.subplot(1, 3, 2)
plt.title('O: %.02f, %s' % (pred_matched, matched_y))
plt.imshow(matched_x.squeeze(), cmap='gray')
plt.subplot(1, 3, 3)
plt.title('X: %.02f, %s' % (pred_unmatched, unmatched_y))
plt.imshow(unmatched_x.squeeze(), cmap='gray')
```

In order to visualize the results obtained from the trained model, the intermediate outputs of convolutional layers from the Siamese CNN are shown in the following figures. Figures 7 and 8 visualize the intermediate outputs from the convolutional layers in the subnetworks when the random fingerprint and the matched fingerprint obtained from the last code snippet are fed forward into the Siamese network as a pair. Figure 9 shows the activations of the rest of the network after the corresponding image feature tensors are obtained from their respective subnetworks.

4 Deep Learning for Character Recognition

4.1 Brief Introduction

Character recognition is the problem of acquiring character in a text and converting it into a digitized format. This involves the detection of characters from paper documents, touch screen devices, scanned documents, etc. It is also one of the earlier tasks since the advent of computer vision. Character recognition techniques are

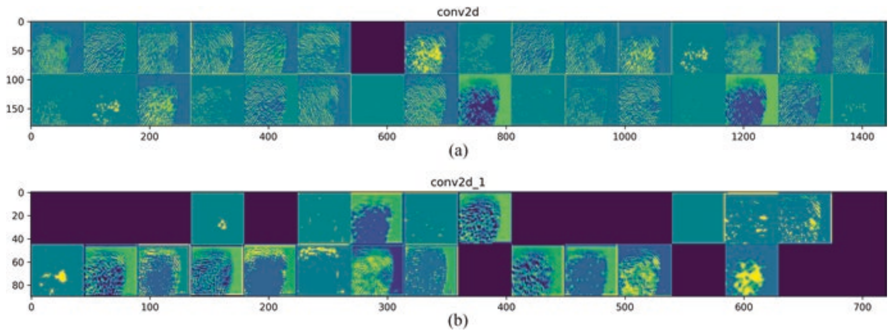


Fig. 7 Intermediate outputs from the convolutional layers in the subnetwork when the random fingerprint chosen is fed forward

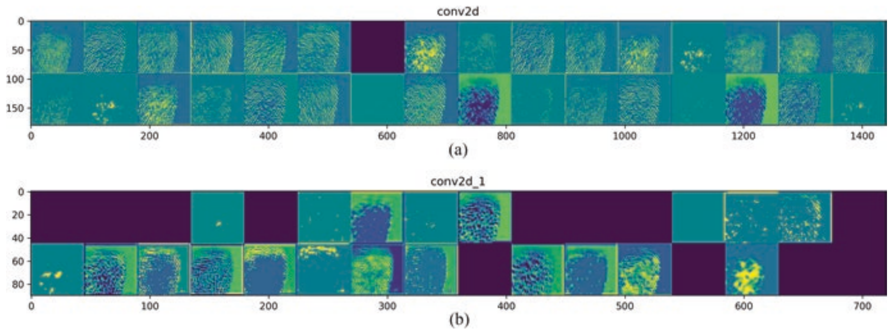


Fig. 8 Intermediate outputs from the convolutional layers in the subnetwork when the matched fingerprint is fed forward

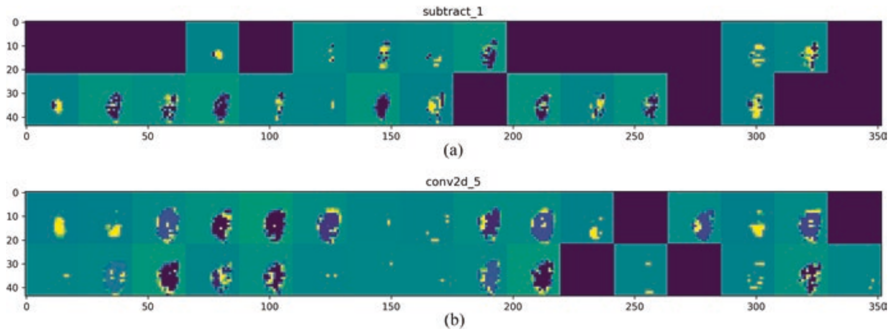


Fig. 9 (**a**) Activation from the subtract layer after the corresponding image feature tensors are obtained from the subnetworks (**b**) Intermediate output from the convolution layer following the subtract layer

widely used along with banking, legal, and other industries. Once the detected characters are converted to machine-readable form, you can export the data into supported file formats for editing or processing the collected data for further use. Even though several works have been able to achieve admirable results, this recognition task remains one of the most challenging due to the presence of its many variations like different writing styles, noise, and other factors.

The character recognition problem can be broken down into its subdomains, digit recognition, and letter recognition. The combination of machine learning approaches like support vector machines (SVM), random forests (RF), k nearest neighbors (kNN), and decision trees (DT), combined with image classification techniques, rooted for the progress of character recognition in the past decade.

These traditional machine learning-based works often provide limited precision in regard to pattern recognition tasks. Deep learning models, since its advent, especially CNNs have been able to alleviate these shortcomings and produce better accurate solutions. The ability of CNNs to be able to extract highly representative image features without the manual tuning of the network makes them highly desirable for character recognition tasks. This has led to a substantial increase in research works trying to incorporate deep learning models to tackle character recognition for various languages [1, 7] and their efforts have been able to surpass the results of former iterations.

4.2 Datasets

The character recognition problem has a substantial number of good benchmarks for analyzing various approaches and techniques. The most popular among them would be the MNIST [27, p. 2287] dataset. The MNIST dataset, which is a subset of the database known as the NIST (National Institute of Standards and Technology) Special Database 19, is an easily accessible public dataset consisting of 60,000 gray scale images of handwritten digits from 0 to 9. Because of the uncomplicated nature of the dataset, most deep learning models and CNNs have been able to achieve incredibly high accuracy on it.

The EMNIST [8, pp.2922–2924] dataset is another dataset derived from the NIST Special Database 19. This dataset directly matches the image specifications and dataset structure of the MNIST dataset. However, the dataset is much more challenging in terms of the interclass similarities and the larger number of output classes. Also, the dataset contains various subsets where each one addresses a different challenge for the researchers. Some examples from the EMNIST dataset are plotted in Fig. 10.

There are also several other datasets used for recognition based upon different languages. One of the most exhaustively used regional datasets is the Devanagari Handwritten Character Dataset [1], a dataset comprising about 1800 handwritten Devanagari characters. Other language datasets like Chars74K [10] and Chinese characters [44] are also extensively used for recognition tasks.

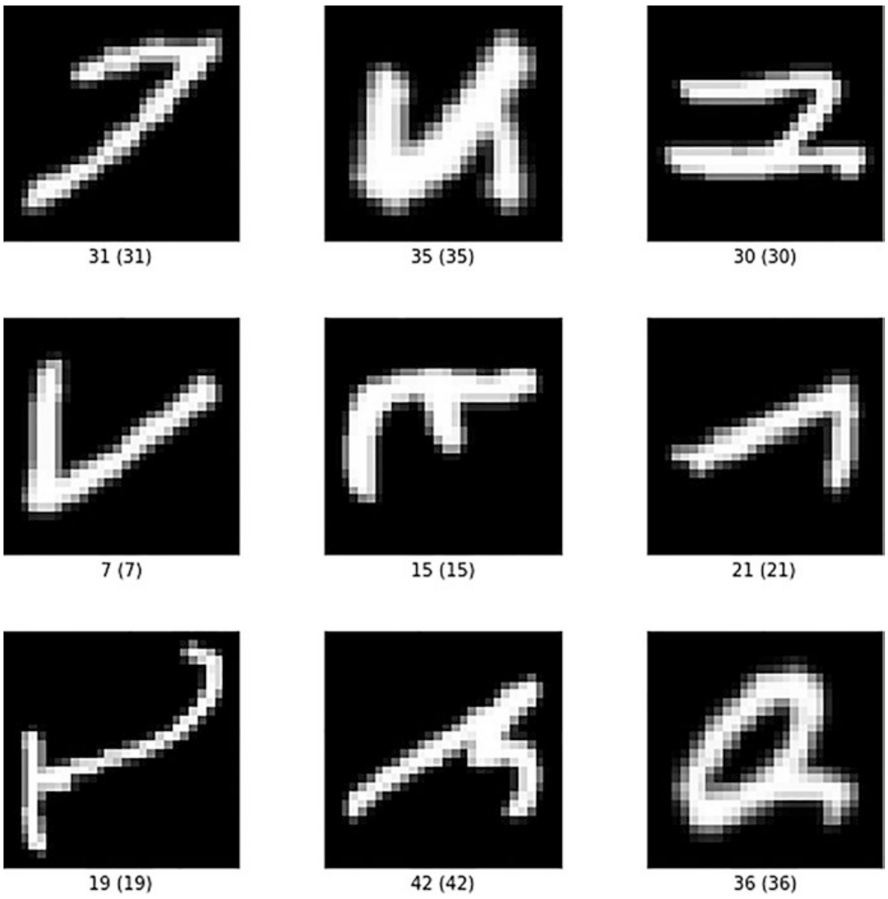


Fig. 10 Random examples from the EMNIST balanced dataset

4.3 Practical Example

As mentioned in the prior subsection, the entire EMNIST dataset is composed of various subsets to address different problems associated with the dataset. In this example, the EMNIST balanced dataset is used to perform character recognition. This 47-class dataset has a fair subset of all the different classes, unlike the complete EMNIST dataset. The following code snippet imports all the necessary libraries. The TensorFlow dataset library enables users to load a collection of public research datasets into the program with ease without the hassle of writing a different script. This package is used to load the balanced version of the EMNIST dataset onto the program.

```
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
import random
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers
from tensorflow.keras import layers
```

Now the **tensorflow_datsasets** package is used to load the dataset onto the program. The **tfds.load()** function downloads and prepares the dataset from the source and loads it to the program. The details of the package and its various helper functions can be found in the official TensorFlow documentation.

```
# load the balanced emnist dataset using tfds.load()
ds, info = tfds.load("emnist/balanced", with_info = True)
# creating objects of train and test datasets
emnist_train, emnist_test = ds["train"], ds["test"]
# setting up the training data
X_train = []
Y_train = []
for example in emnist_train.as_numpy_iterator():
 image, label = example['image'],example['label']
 X_train.append(image)
 Y_train.append(label)
# setting up the test data
X_test = []
Y_test = []
for example in emnist_test.as_numpy_iterator():
 image, label = example['image'],example['label']
 X_test.append(image)
 Y_test.append(label)
```

Now that the dataset has been loaded into separate training and test sets, preprocessing of these has to be conducted. Since the pixel values range from 0 to 255, they are converted to the range of 0–1 by dividing the whole NumPy array by 255. This maneuver will further produce a reduction in the magnitude of computation. Also, all the EMNIST images by default are inverted horizontally and rotated by 90 degrees. So the **np.transpose()** operation has to be done on all the pictures in both the training and test set to set them straight. The result of these operations can be seen in Fig. 11.

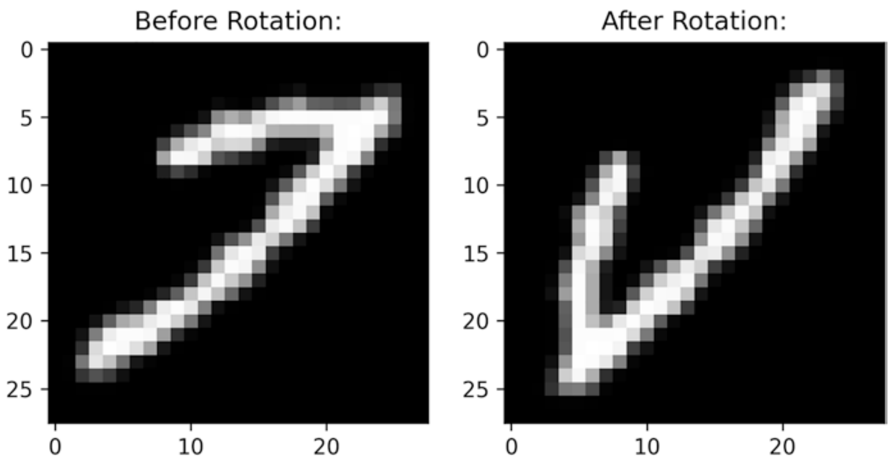


Fig. 11 Before and after inversion and rotation

```
X_train, X_test = np.array(X_train) / 255. , np.array(X_test) / 255.
X_train = X_train.reshape(len(X_train), 28, 28)
X_test = X_test.reshape(len(X_test), 28, 28)
train_images = X_train.astype('float32')
test_images = X_test.astype('float32')
# plot the image before rotation
f = plt.figure(figsize=(7,7))
f.add_subplot(1,2,1)
plt.title("Before Rotation:")
plt.imshow(train_images[0], cmap='gray')
# To rotate and reverse these images
# for train data
for i in range(train_images.shape[0]):
   train_images[i] = np.transpose(train_images[i])
# for test data
for i in range(test_images.shape[0]):
   test_images[i] = np.transpose(test_images[i])
# plot the image after rotation
f.add_subplot(1,2,2)
plt.imshow(train_images[0], cmap='gray')
plt.title("After Rotation:")
# show the plot
plt.show()
```

Also, the label must be preprocessed in such a way that the label array must be converted into a binary class matrix. This can be performed using the **tf.keras.utils.to_categorical** function which will return a binary matrix representation of the input.

```
train_labels = tf.keras.utils.to_categorical(Y_train, 47)
test_labels = tf.keras,utils.to_categorical(Y_test, 47)
```

The next step is to model the CNN required for this example. Since the EMNIST dataset is more complicated than the MNIST dataset due to the intrinsic variations, a deeper CNN will be necessary to capture the features from them for better prediction. The network is modeled using the Sequential model provided by Keras. Since the dataset comprises multiple classes, the **categorical_crossentropy** loss function, along with the **Adamax** optimizer, will be used to compile the model. The following lines of code will model the network for this example.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Convolution2D as Conv2D
from tensorflow.keras.layers import BatchNormalization, Dropout, Flatten, Dense, Reshape
from tensorflow.keras import optimizers
model = Sequential()
model.add(Conv2D(32, kernel_size = 3, activation='relu', input_shape = (28, 28, 1)))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size = 5, strides=2, padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Conv2D(64, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size = 5, strides=2, padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Conv2D(128, kernel_size = 4, activation='relu'))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(47, activation='softmax'))

opt = optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Once the model is defined, it can be trained using the **model.fit()** function. The training and testing images, along with their labels, are passed to the function. The batch size, along with the number of epochs for training, is also specified as arguments.

```
history = model.fit(train_images,train_labels,validation_data=(test_images, test_labels), batch_size=128, epochs=20)
```

The model on training for 20 epochs attains a training accuracy of 90.03% and a validation accuracy of 89.78%. The training and validation accuracy curves are plotted using the history callback returned. The resultant graphs produced are shown in Fig. 12.

```
f = plt.figure(figsize=(12,5))
f.add_subplot(1,2,1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
f.add_subplot(1,2,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

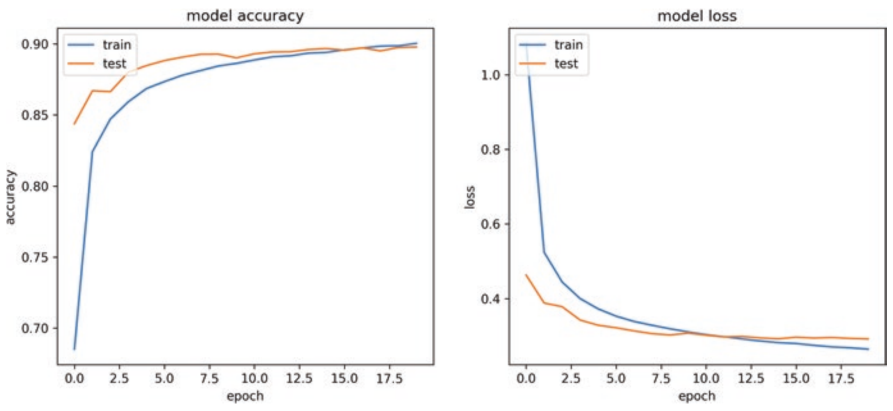


Fig. 12 Training and validation accuracy and loss curves

Now that the model is trained, testing can be conducted by taking a random image from the pile of test images and feeding it forward through the network. The index of a random image is chosen by using the randrange function provided by the random module. The **model .predict()** function is used to produce the prediction of the network. The **np.argmax()** function is used to obtain the index of the class with the largest probability.

```
random_idx = random.randrange(0, len(test_images))
test_img = np.expand_dims(test_images[random_idx], 0)
result = model.predict(test_img)
result = np.argmax(result, axis=1)
test_img = np.reshape(test_img, (28,28))
plt.imshow(test_img, cmap='gray')
plt.title("Predicted: " + class_mapping[result[0]])
plt.show()
```

The random image, along with the predicted class label, is plotted using the **matplotlib** library in the above code fragment. The result is shown in Fig. 13.

Figure 14 shows the intermediate outputs produced by the first three convolutional layers in the network on feed forwarding the random image that was chosen in the last code snippet. It is observed that the input image features become much less abstract as they move deeper into the CNN.

5 Deep Learning for Smart Grids

5.1 Brief Introduction

Smart grid technology evolved from a system of over-a-century-old conventional grids. While standard grids can only provide one-way communication from the generation to the consumer, smart grids can provide two-way communication. The

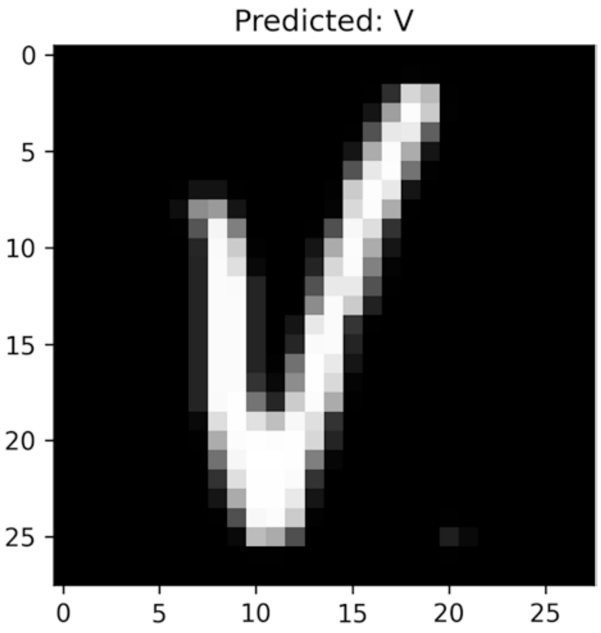


Fig. 13 Random image chosen for prediction along with the predicted class label

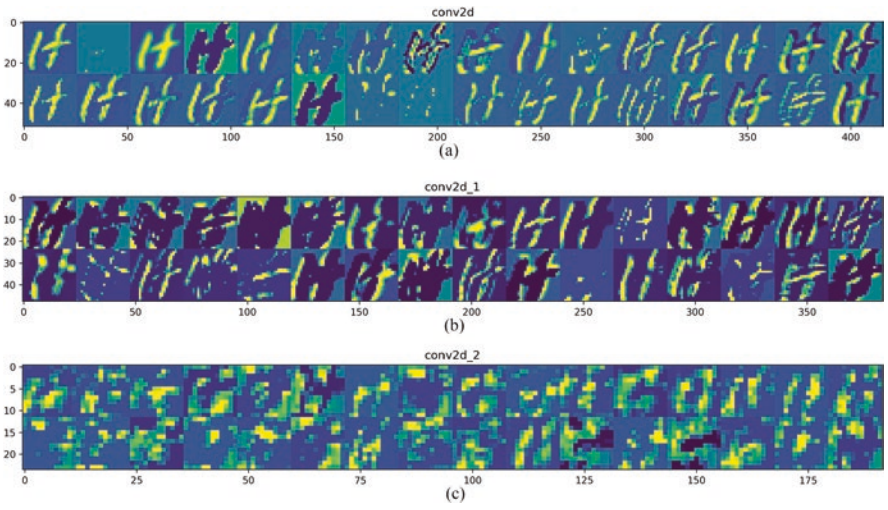


Fig. 14 (a–c) Intermediate outputs from the first three convolutional layers on feed forwarding the random image chosen earlier for testing communication network can also aggregate the actions of all the users and efficiently provide the whole system with lower losses, high quality, and more reliability, along with environmental sustainability and economic efficiency. The arrival of smart grids caused an increase in the use of digital information, more control features, and more requirements for the grid system analytics.

Deep learning has had significant success in load forecasting, microgrid management, demand response, fault analysis, etc. in a smart grid. Application-oriented deep learning systems can be provided for these problems, thus, achieving more accuracy and reliability. One of the most popular terms related to smart grids is smart metering. Smart metering is the technology that records consumption of electric energy, and the information is updated to the supplier on a regular basis. Typical smart meters record energy on an hourly or more frequent basis and report at least daily. Deep learning techniques can be applied to forecast the energy consumption by smart meters in residential households by taking into consideration the weather, consumer activity pattern, and other statistics.

Early works in the application of deep learning in smart grids used to include either one of CNNs(Convolutional Neural Networks), K-means, LSTMs (Long Short Term Memory), GRUs(Gated Recurrent Units) or a combination of these for the forecasting of hourly or daily loads [30, 43]. The addition of renewable resources and customer participation results in an uncertain and complex environment. To ensure secure and stable operation of power systems, DRL (deep reinforcement learning) can be used for computing accurate control schemes for decision and control problems [31, 41].

5.2 Datasets

One of the most accessible datasets for smart grids is the smart meters in London data (“Smart Meters In London”, 2020), which contains the energy consumption readings for a sample of 5567 households in London City which contributed to the Low Carbon London Project during the period from November 2011 to February 2014. The smart meter data is correlated only with electricity consumption.

Other popular datasets include the Residential Energy Consumption Survey dataset [40], which contains energy characteristics on the housing unit, usage patterns, and household demographics. Another one is the Reference Energy Disaggregation dataset [25], which consists of power data collected across several weeks among six different homes and main power supply characteristics like high-frequency current/voltage data for two of these homes.

5.3 Practical Example

In this example, a deep learning model is used to predict the average energy consumption in kWh from a time series data. The model is trained using the dataset obtained from smart meters in London mentioned in the prior subsection. This dataset contains information regarding energy consumption per household, across various time durations. The data also includes information on the hourly and daily weather conditions, and this must be taken into consideration for the preparation of the data frame.

Energy consumption data is taken per day per household. Doing so helps normalize data for the household count, which tends to be inconsistent across the dataset. The relationships between weather conditions and energy consumption are explored by creating clusters for the weather data and then adding weather identifiers to day-level data. The UK holiday data is added as an indicator to the day-level data, and the time series data is then predicted using a GRU (gated recurrent unit).

Initially, all the necessary libraries are imported.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import math
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import GRU
```

The predictions are made for energy demand in the future. Therefore only the energy sum, i.e., total energy use per day for a given household, is taken. The data is prepared according to these needs from the entire dataset using the following code snippet.

```
for num in range(0,112):
    df = pd.read_csv("../input/daily_dataset/daily_dataset/block_"+str(num)+".csv")
    df = df[['day','LCLid','energy_sum']]
    df.reset_index()
    df.to_csv("hc_"+str(num)+".csv")

fout= open("energy.csv","a")
# first file:
for line in open("hc_0.csv"):
    fout.write(line)
# now the rest:
for num in range(0,112):
    f = open("hc_"+str(num)+".csv")
    f.readline() # skip the header
    for line in f:
          fout.write(line)
    f.close()
fout.close()
```

The code will create a file named "**energy.csv**" with four columns: index, day, LC Lid, and energy sum. It can be observed from the dataset that the energy data collected across different days have different numbers of households. This observation can be due to the phenomenon of the increased acquiring of smart meters in London. It could lead to a false interpretation that the energy for a particular day might be high when there could be a chance that the data was only collected for more houses. For verifying this observation, house count for each day is observed.

```
housecount = energy.groupby('day')[['LCLid']].nunique()
housecount.head(4)

>>>
LCLid         day
2011-11-23    13
2011-11-24    25
2011-11-25    32
2011-11-26    41
```

The data collection across households is inconsistent. Therefore "energy per household" will be the target to predict rather than energy alone.

```
energy = energy.groupby('day')[['energy_sum']].sum()
energy = energy.merge(housecount, on = ['day'])
energy = energy.reset_index()
energy.count()

>>> day           829
    energy_sum    829
    LCLid         829
    dtype: int64
```

The daily average energy is now obtained:

```
energy.day = pd.to_datetime(energy.day,format='%Y-%m-%d').dt.date
energy['avg_energy'] =  energy['energy_sum']/energy['LCLid']
```

The daily level of weather information containing 32 columns is taken using the dark sky API in the dataset. Only the relevant attributes from the weather data are chosen, and any rows containing null values are also dropped.

```
weather = pd.read_csv('weather_daily_darksky.csv')
weather.head(4)
weather['day']=  pd.to_datetime(weather['time']) # day is given as timestamp
weather['day']=  pd.to_datetime(weather['day'],format='%Y%m%d').dt.date
# selecting numeric variables
weather = weather[['temperatureMax', 'windBearing', 'dewPoint', 'cloudCover', 'windSpeed',
       'pressure', 'apparentTemperatureHigh', 'visibility', 'humidity',
       'apparentTemperatureLow', 'apparentTemperatureMax', 'uvIndex',
       'temperatureLow', 'temperatureMin', 'temperatureHigh',
       'apparentTemperatureMin', 'moonPhase','day']]
weather = weather.dropna()
```

The weather data is now merged with energy data to analyze the relationship between weather conditions and electricity consumption.

```
weather_energy =  energy.merge(weather,on='day')
```

The correlation matrix contains the columns of the weather variables, which has a significant correlation with energy consumption.

```
cor_matrix = weather_energy[['avg_energy','temperatureMax','dewPoint', 'cloudCover', 'windSpeed','pressure',
'visibility', 'humidity','uvIndex', 'moonPhase']].corr()
```

Weather clusters have to be created to see if the weather of the day can be defined based on features like temperature and precipitation and avoid unnecessary information in the weather data. Before creating clusters, the values must be normalized into the feature range of 0–1. The normalization is achieved using the **MinMaxScaler()** provided by the **sklearn** library. Once the values are normalized, the clusters can be created using the **KMeans** function, also supplied by the same library.

```
weather_scaled = scaler.fit_transform(weather_energy[['temperatureMax','humidity','windSpeed']])
kmeans = KMeans(n_clusters=3, max_iter=600, algorithm = 'auto')
kmeans.fit(weather_scaled)
weather_energy['weather_cluster'] = kmeans.labels_
```

The UK Bank holidays data is now loaded to set as an indicator.

```
holiday = pd.read_csv('uk_bank_holidays.csv')
holiday['Bank holidays'] = pd.to_datetime(holiday['Bank holidays'],format='%Y-%m-%d').dt.date
holiday.head(4)
```

```
>>>          Bank holidays                              Type
0    2012-12-26                                  Boxing Day
1    2012-12-25                               Christmas Day
2    2012-08-27                         Summer bank holiday
3    2012-05-06  Queen?s Diamond Jubilee (extra bank holiday)
```

A holiday indicator is now created in the weather energy data, and values of the series data are obtained from the data frame, as shown below.

```
weather_energy = weather_energy.merge(holiday, left_on = 'day',right_on = 'Bank holidays',how = 'left')
weather_energy['holiday_ind'] = np.where(weather_energy['Bank holidays'].isna(),0,1)
dataframe = weather_energy.loc[:,'avg_energy']
dataset = dataframe.values
dataset = dataset.astype('float32')
```

The series of values need to be transformed into a supervised learning problem for passing into the deep learning model for prediction. This could be done by taking a group of samples from the rows and tabulating them into columns, and the value in the last column is set as the target to be predicted. The "convert_to_supervised" function is defined and applied to the dataset, and the first five rows of the supervised data are displayed.

```
def convert_to_supervised(data, n_in=1, n_out=1, dropnan=True):

    columns, names = list(), list()
    df = pd.DataFrame(data)
    n_var = 1
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        columns.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_var)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        columns.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_var)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_var)]
    # put it all together
    con = pd.concat(columns, axis=1)
    con.columns = names
    # drop rows with NaN values
    if dropnan:
        con.dropna(inplace=True)
    return con


supervised = convert_to_supervised(dataset, 7,1)
supervised.head()

>>>     var1(t-7)  var1(t-6)  var1(t-5)  ...  var1(t-2)  var1(t-1)    var1(t)
7    6.952693   8.536480   9.499782  ...   9.103382   9.274873   8.813513
8    8.536480   9.499782  10.267707  ...   9.274873   8.813513   9.227707
9    9.499782  10.267707  10.850805  ...   8.813513   9.227707  10.145910
10  10.267707  10.850805   9.103382  ...   9.227707  10.145910  10.862155
11  10.850805   9.103382   9.274873  ...  10.145910  10.862155  12.351882
```

After reindexing the supervised data with weather cluster data and holiday index, the data frame is converted to a series of values.

```
supervised['weather_cluster'] = weather_energy.weather_cluster.values[7:]
supervised['holiday_ind']= weather_energy.holiday_ind.values[7:]
supervised = supervised.reindex(['weather_cluster', 'holiday_ind','var1(t-7)', 'var1(t-6)', 'var1(t-5)',
'var1(t-4)', 'var1(t-3)','var1(t-2)', 'var1(t-1)', 'var1(t)'], axis=1)
supervised = supervised.values
```

The data is now split into train and test datasets using sklearn's train_test_split function following the transformation of the data to a feature range between 0 and 1 using the **MinMaxScaler** function.

```
scaler = MinMaxScaler(feature_range=(0, 1))
supervised = scaler.fit_transform(supervised)
# split into train and test sets

train_X,test_X,train_y,test_y=train_test_split(supervised[:,:-1],supervised[:,-1], test_size=0.3,shuffle=False)
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
```

The model architecture is defined in the following steps. The input is passed through GRU (gated recurrent units) layer after passing through the input layer and then finally through a dense layer. The loss function used here is MSE (mean squared error) and is popularly used for time series applications.

```
model = Sequential()
model.add(GRU(50, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Now the data is trained over the model for 50 epochs, with a batch size of 72. The training loss is calculated from the training set and plotted using the **pyplot** module of **matplotlib**. The result is shown in Fig. 15

```
history = model.fit(train_X, train_y, epochs=50, batch_size=72,verbose=1, shuffle=False)
plt.figure(1)
plt.plot(history.history['loss'], label='train loss')
plt.xlabel('epochs', fontsize=14)
plt.ylabel('loss', fontsize=14)
plt.legend()
plt.show()
```

The predictions are made as shown, and RMSE (root mean square error) is calculated from the actual and predicted values.

```
# make a prediction
transformed_pred = model.predict(test_X)
test_X = test_X.reshape(test_X.shape[0], test_X.shape[2])
# invert scaling for forecast
prediction = np.concatenate((test_X,transformed_pred), axis=1)
prediction = scaler.inverse_transform(prediction)
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
actual_y = np.concatenate(( test_X,test_y), axis=1)
actual_y = scaler.inverse_transform(actual_y)
act = actual_y[:,-1]
pred = prediction[:,-1]

# calculate RMSE
rmse = math.sqrt(mean_squared_error(act, pred))
print('Test RMSE: %.3f' % rmse)


>>> Test RMSE: 0.793
```

The actual average energy and predicted values are plotted as shown in Fig. 16.

```
predicted_gru = pd.DataFrame({'predicted':pred,'avg_energy':act})
plt.figure(2)
plt.plot(predicted_gru['avg_energy'],label='actual')
plt.plot(predicted_gru['predicted'],label='predicted')
plt.xlabel('test sample number', fontsize=14)
plt.ylabel('Energy in kWh', fontsize=14)
plt.legend()
plt.show()
```

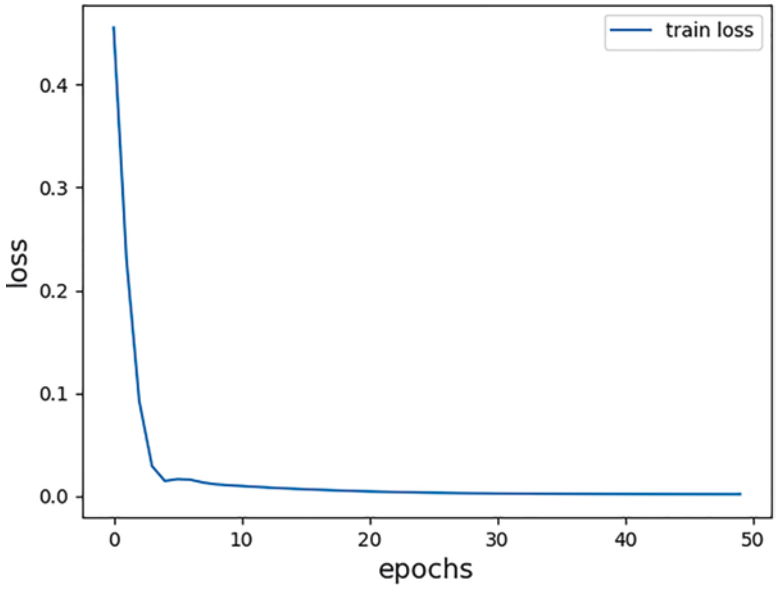


Fig. 15 Training loss curve

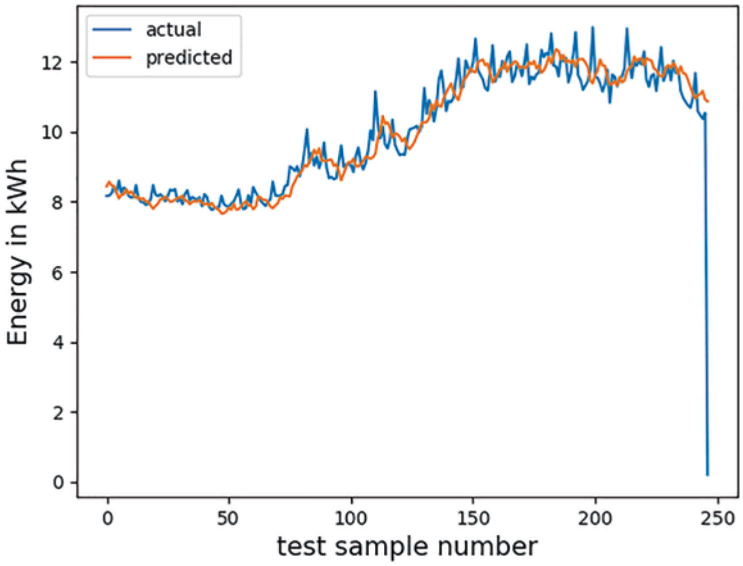


Fig. 16 Actual vs. predicted energy in KwH

6 Deep Learning in Renewable Energy and Sustainable Development

6.1 Brief Introduction

As energy production is needed on a much larger scale, optimizing the usage and output of currently available energy sources is a big concern. Solar and wind energy, being the two primary sources of renewable energy, are variable. The challenges in forming an optimal renewable energy system are the instability and the limit in power output. Deep learning these days is extensively used in the forecasting of production, usage, etc. in the energy system.

One of the widely solved problems in the renewable energy domain using deep learning is forecasting. The typical machine learning methods involved for forecasting are done with the help of linear regression techniques and artificial neural networks (ANN). Such techniques include the usage of a lot of statistical aspects, including long short-term memory (LSTM), gated recurrent unit (GRU), AutoEncoder LSTM (Auto-LSTM), and a newly proposed technique known as Auto-GRU [3, 13]. The LSTM-based approach is a modification to RNNs (recurrent neural networks) to help resolve the vanishing gradient problem. LSTM has a memory cell that allows it to store information in memory for a long time, thus facilitating the learning process in the feedback loops.

While forecasting renewable energy, a wide variety of factors and approaches can be taken into consideration. Early work in forecasting the wind energy has been done by predicting the wind speed and mapping to wind power using given power curves [46]. This involves predicting the rate of different wind turbines situated on wind farms. While considering solar energy, forecasting can be done by using the knowledge of the Sun’s path, the atmospheric conditions, the scattering processes, and other properties of the photovoltaic cells used in the solar farm to capture energy [17]. The output is highly dependent on the climate, and summer seasons are likely to attract more solar energy. Also, the abundance of this resource is only during a few hours of the day.

6.2 Datasets

The California renewable energy production dataset (“California Renewable Production 2010-2018”, 2020) consists of hourly metering of the power production in California between 2010 and 2018. It consists of the power reports from various power sources on a megawatt-scale. Familiar sources include geothermal, small hydro, biomass, biogas, wind, solar PV (photovoltaics), and solar thermal. The source of the data is from the California ISO renewable and emission reports [4].

The International Energy Statistics (International Energy Statistics, 2020) provides energy statistics on various levels, and the data about the new and renewable sources of energy, like wind and solar energy, can be extracted from it. This dataset was obtained by the UN Statistics Division [39].

6.3 Practical Example

In this section, a deep learning model is trained to predict the solar photovoltaic output. A univariate time series prediction is made using LSTM [47]. The model is trained using California Renewable Production 2010–2018 dataset. It consists of hourly metered power production details from various power sources in a CSV (comma-separated values) format. Among these, the solar photovoltaic data is considered. The data is converted to rows containing batches of the series, and the model learns to predict the next term in the series [48]. We use MSE (mean square error), which is the most common loss function used for regression problems. MSE calculates the sum of the square of the distance between the actual and target values [49].

The necessary libraries are imported. **Pandas** is an open-source python library which is useful in data analysis and manipulation tasks. The **sklearn** library is used to do a feature transformation to normalize the data values in a range of 0–1 [50].

```
import pandas as pd
from pandas import DataFrame
from pandas import concat
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot as plt
from math import sqrt
from numpy import concatenate
```

After importing the necessary libraries, the data is loaded using the **read_csv** function. The "usecols" parameter here takes an array of only the required columns to be taken into the data frame. The obtained data is now sorted according to timestamp and is filtered by **notna()** function to drop the rows containing NaN or null values. The first five rows of the data frame can be obtained using the **head()** function.

```
data = pd.read_csv('all_breakdown.csv',usecols=["TIMESTAMP","SOLAR PV"])
data=data.sort_values(by=['TIMESTAMP'])
df = data[data['SOLAR PV'].notna()]
print(df.head())

>>>                TIMESTAMP  SOLAR PV
35280  2012-11-26 00:00:00       0.0
35281  2012-11-26 01:00:00       0.0
35282  2012-11-26 02:00:00       0.0
35283  2012-11-26 03:00:00       0.0
35284  2012-11-26 04:00:00       0.0
```

The value property returns the values in the data frame. The column containing the univariate time series data is retrieved and ensured if all the data is of the same type float32, using the astype function. After this, the values are normalized into features ranging from 0 to 1.

```
values = df.values[:,1:]
# ensure all data is float
values = values.astype('float32')
# normalize features
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
```

Before training the model, preprocessing is done to the data to convert the series data to supervised data. This process is done by taking a particular group of samples and tabulating all of them in the same row so that the last column is the sample that needs to be predicted. The normalized data obtained is then passed on to this function to return a data frame in the form which enables supervised learning.

```
def convert_to_supervised(data, n_in=1, n_out=1, dropnan=True):

    columns, names = list(), list()
    df = pd.DataFrame(data)
    n_var = 1
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        columns.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_var)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        columns.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_var)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_var)]
    # put it all together
    con = pd.concat(columns, axis=1)
    con.columns = names
    # drop rows with NaN values
    if dropnan:
        con.dropna(inplace=True)
    return con


supervised = convert_to_supervised(scaled, 2, 2)
print(supervised.head())

The result of the above function is shown by displaying the first five entries of the data in supervised form.

   var1(t-2)  var1(t-1)  var1(t)  var1(t+1)
2        0.0        0.0  0.00000   0.000000
3        0.0        0.0  0.00000   0.000000
4        0.0        0.0  0.00000   0.000000
5        0.0        0.0  0.00000   0.000960
6        0.0        0.0  0.00096   0.016964
```

Now that the data is prepared in the form desired, the train and test set splitting is done using **train_test_split** function. Both the train and test sets are reshaped for matching the size for training parameters.

```
train=supervised.values
train_X,test_X,train_y,test_y=train_test_split(train[:,:-1],train[:,-1], test_size=0.3)
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
```

The model architecture can now be defined. An LSTM layer followed with two dense layers is added. Mean square error is the preferred and most popular loss function for regression problems. The model is then run for 50 epochs with a batch size of 20.

```
model = Sequential()
model.add(LSTM(200,input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dense(64))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
# fit network
history = model.fit(train_X, train_y, epochs=50, batch_size=20, verbose=1, shuffle=False)
```

At the end of 50 epochs, the training loss is in the range of $9.3 * 10^{-4}$. The history callback returned by the **model.fit()** function is used to plot the training loss curve. The callback values are plotted using the **pyplot** function of the **matplotlib** library. The resultant loss curve is shown in Fig. 17.

```
plt.figure(1)
plt.plot(history.history['loss'], label='train')
plt.xlabel('epochs', fontsize=14)
plt.ylabel('loss', fontsize=14)
plt.legend()
plt.show()
```

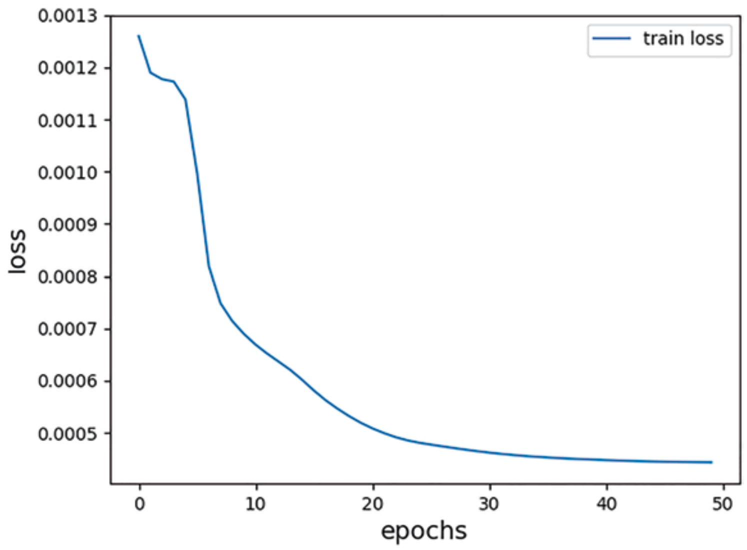


Fig. 17 Training loss curve

The model can now predict the test set using the **model.predict()** function, and this returns a list containing all the predicted values. The predicted and actual values are passed on to the **inverse transform** function to convert them back from their normalized values to real-world data. The RMSE is calculated to analyze how well the model performed.

```
transformed_pred = model.predict(test_X)
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
# invert scaling for forecast
prediction = concatenate((transformed_pred, test_X[:, 1:]), axis=1)
prediction = scaler.inverse_transform(prediction)
```

```
# make a prediction
transformed_pred = model.predict(test_X)
test_X = test_X.reshape(test_X.shape[0], test_X.shape[2])
# invert scaling for forecast
prediction = np.concatenate((test_X,transformed_pred), axis=1)
prediction = scaler.inverse_transform(prediction)
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
actual_y = np.concatenate(( test_X,test_y), axis=1)
actual_y = scaler.inverse_transform(actual_y)
act = actual_y[:,-1]
pred = prediction[:,-1]

# calculate RMSE
rmse = math.sqrt(mean_squared_error(act, pred))
print('Test RMSE: %.3f' % rmse)

>>> Test RMSE: 469.556
```

A plot between the predicted and actual power values is plotted using the **pyplot** function to visualize the results which is shown in Fig. 18.

```
plt.figure(2)
plt.plot(inv_y[:50], label='actual')
plt.plot(inv_yhat[:50], label='predicted')
plt.xlabel('test sample number', fontsize=14)
plt.ylabel('power in MW', fontsize=14)
plt.legend()
plt.show()
```

7 Conclusion

The applications of deep learning extend to many aspects of daily life and are not confined to the domains of computer science alone. From face recognition to the smart grid domain, deep learning has proved itself to be an effective tool. In this chapter, we have discussed how the usage of a high-level language like Python and its compatibility with deep learning frameworks and its collection of utility libraries

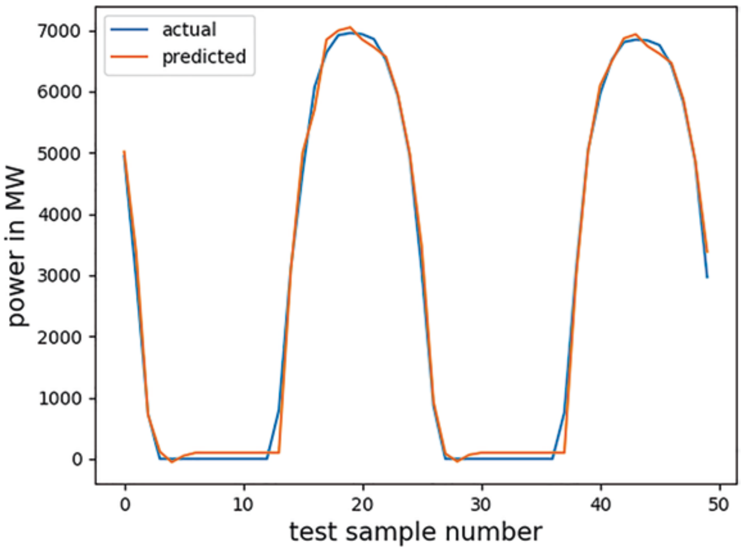


Fig. 18 Actual vs predicted power in MW

facilitates practitioners in the development process. We also saw how deep learning has affected different applications and domains and walked through a complete Python implementation comprehensively covering the different steps in creating a solution in each subsection.

References

1. Acharya, S., Pant, A.K., Gyawali, P.K.: Deep learning based large scale handwritten Devanagari character recognition. In *2015 9th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)* (pp. 1-6). IEEE (2015, December)
2. Ahonen, T., Hadid, A., Pietikainen, M.: Face description with local binary patterns: application to face recognition. IEEE Trans. Pattern Anal. Mach. Intell. **28**(12), 2037–2041 (2006)
3. Aslam, M., Lee, J.M., Kim, H.S., Lee, S.J., Hong, S.: Deep learning models for long-term solar radiation forecasting considering microgrid installation: a comparative study. *Energies*. ***13***(1), 147 (2020)
4. California ISO, California ISO – Renewables and emissions reports, caiso.com, http://www.caiso.com/market/Pages/ReportsBulletins/RenewablesReporting.aspx (accessed May 17, 2020)
5. Cao, Q., Shen, L., Xie, W., Parkhi, O.M., Zisserman, A.: Vggface2: a dataset for recognising faces across pose and age. In *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)* (pp. 67-74). IEEE (2018, May)
6. Centeno, Iván de Paz. 2020. "Mtcnn". *Pypi*. https://pypi.org/project/mtcnn/
7. Chen, L., Wang, S., Fan, W., Sun, J., Naoi, S.: Beyond human recognition: a CNN-based framework for handwritten character recognition. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)* (pp. 695–699). IEEE (2015, November)

8. Cohen, G., Afshar, S., Tapson, J., Van Schaik, A.: EMNIST: extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)* (pp. 2921–2926). IEEE (2017, May).
9. Darlow, L.N., Rosman, B.: Fingerprint minutiae extraction using deep learning. In *2017 IEEE International Joint Conference on Biometrics (IJCB)* (pp. 22–30). IEEE (2017, October)
10. De Campos, T., Bodla, R. B., Varma, M.: The chars74k dataset (2009)
11. FVC-onGoing 2009 On-line evaluation of fingerprint recognition algorithms, https://biolab.csr.unibo.it/fvcongoing
12. Garcia-Salicetti, S., Beumier, C., Chollet, G., Dorizzi, B., Les Jardins, J.L., Lunter, J., Ni, Y., Petrovska-Delacrétaz, D.: BIOMET: a multimodal person authentication database including face, voice, fingerprint, hand and signature modalities. In: *International conference on audio- and video-based biometric person authentication*, pp. 845–853. Springer, Berlin, Heidelberg (2003, June)
13. Gensler, A., Henze, J., Sick, B., Raabe, N.: Deep learning for solar power forecasting—an approach using AutoEncoder and LSTM neural networks. In *2016 IEEE international conference on systems, man, and cybernetics (SMC)* (pp. 002858–002865). IEEE (2016, October)
14. Guo, Y., Zhang, L., Hu, Y., He, X., Gao, J.: Ms-celeb-1m: a dataset and benchmark for large-scale face recognition. In: *European conference on computer vision*, pp. 87–102. Springer, Cham (2016, October)
15. 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)
16. Huang, G.B., Mattar, M., Berg, T., Learned-Miller, E.: Labeled faces in the wild: A database forstudying face recognition in unconstrained environments (2008, October)
17. Inman, R.H., Pedro, H.T., Coimbra, C.F.: Solar forecasting methods for renewable energy integration. Prog. Energy Combust. Sci. **39**(6), 535–576 (2013)
18. Jain, A.K., Li, S.Z.: *Handbook of face recognition*, vol. 1. Springer, New York (2011)
19. Jiang, B., Ren, Q., Dai, F., Xiong, J., Yang, J., Gui, G.: Multi-task cascaded convolutional neural networks for real-time dynamic face recognition method. In: *International conference in communications, signal processing, and systems*, pp. 59–66. Springer, Singapore (2018, July)
20. Kaggle, "California Renewable Production 2010–2018", kaggle.com, https://www.kaggle.com/cheedcheed/california-renewable-production-20102018 (accessed May 17, 2020)
21. Kaggle, "Smart meters in London", kaggle.com, https://www.kaggle.com/jeanmidev/smart-meters-in-london/ (accessed May 17, 2020)
22. Kaggle, "International Energy Statistics", kaggle.com, https://www.kaggle.com/unitednations/international-energy-statistics (accessed May 17, 2020)
23. Kemelmacher-Shlizerman, I., Seitz, S.M., Miller, D., Brossard, E.: The megaface benchmark: 1 million faces for recognition at scale. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4873–4882) (2016)
24. Koch, G., Zemel, R., Salakhutdinov, R.: Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop* (Vol. 2) (2015, July)
25. Kolter, J.Z., Johnson, M.J.: REDD: a public data set for energy disaggregation research. In: *Workshop on data mining applications in sustainability (SIGKDD)*, vol. 25, pp. 59–62. Citeseer, *San Diego, CA* (2011, August)
26. 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)
27. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE. **86**(11), 2278–2324 (1998)
28. Liu, C., Wechsler, H.: Gabor feature based classification using the enhanced fisher linear discriminant model for face recognition. IEEE Trans. Image Process. **11**(4), 467–476 (2002)
29. Malli, R., 2020. *Keras-Vggface*. [online] PyPI. Available at: https://pypi.org/project/keras-vggface [Accessed 17 May 2020]

30. Marino, D.L., Amarasinghe, K. and Manic, M., 2016, October. Building energy load forecasting using deep neural networks. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society* (pp. 7046–7051). IEEE
31. Mbuwir, B., Ruelens, F., Spiessens, F., Deconinck, G.: Reinforcement learning-based battery energy management in a solar microgrid. *Energy-Open*. ***2***(4), 36 (2017)
32. Ortega-Garcia, J., Fierrez-Aguilar, J., Simon, D., Gonzalez, J., Faundez-Zanuy, M., Espinosa, V., Satue, A., Hernaez, I., Igarza, J.J., Vivaracho, C., Escudero, D.: MCYT baseline corpus: a bimodal biometric database. IEE Proceedings-Vision, Image and Signal Processing. **150**(6), 395–401 (2003)
33. Parkhi, O.M., Vedaldi, A. and Zisserman, A., 2015. Deep face recognition
34. Peralta, D., Triguero, I., García, S., Saeys, Y., Benitez, J.M. and Herrera, F., 2017. Robust classification of different fingerprint copies with deep neural networks for database penetration rate reduction. *arXiv preprint arXiv:1703.07270*
35. Schroff, F., Kalenichenko, D. and Philbin, J., 2015. Facenet: a unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 815-823)
36. Shehu, Y.I., Ruiz-Garcia, A., Palade, V., James, A.: Detection of fingerprint alterations using deep convolutional neural networks. In: *International conference on artificial neural networks*, pp. 51–60. Springer, Cham (2018, October)
37. Taigman, Y., Yang, M., Ranzato, M.A. and Wolf, L., 2014. 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)
38. Turk, M., Pentland, A.: Eigenfaces for recognition. J. Cogn. Neurosci. **3**(1), 71–86 (1991)
39. UNdata, "UNdata | explorer", data.un.org, http://data.un.org/Explorer.aspx (accessed May 17, 2020)
40. US Energy Information Administration (EIA), "Residential Energy Consumption Survey (RECS) – Data – US Energy Information Administration (EIA)", eia.gov, https://www.eia.gov/consumption/residential/data/2015/ (accessed May 17, 2020)
41. Wang, H., Huang, T., Liao, X., Abu-Rub, H., Chen, G.: Reinforcement learning in energy trading game among smart microgrids. IEEE Trans. Ind. Electron. **63**(8), 5109–5119 (2016)
42. Yi, D., Lei, Z., Liao, S. and Li, S.Z., 2014. Learning face representation from scratch. *arXiv preprint arXiv:1411.7923*
43. Zhang, D., Han, X., Deng, C.: Review on the research and practice of deep learning and reinforcement learning in smart grids. CSEE Journal of Power and Energy Systems. **4**(3), 362–370 (2018)
44. Zhou, S., Chen, Q. and Wang, X., 2010, June. HIT-OR3C: an opening recognition corpus for Chinese characters. In *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems* (pp. 223–230)
45. Shrein, John M. "Fingerprint classification using convolutional neural networks and ridge orientation images." In 2017 IEEE Symposium Series on Computational Intelligence (SSCI), (pp. 1–8) IEEE (2017).
46. Ghaderi, Amir, Borhan M. Sanandaji, and Faezeh Ghaderi. "Deep forecast: Deep learning-based spatio-temporal forecasting." arXiv preprint arXiv:1707.08110 (2017)
47. JHA, A. K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor Flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
48. Kanagachidambaresan, G. R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
49. Nagapawan, Y.V.R., Prakash, K.B., Kanagachidambaresan, G.R.: Convolution neural network, programming with tensor Flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_6
50. Kanagachidambaresan, G.R., Ruwali, A., Banerjee, D., Prakash, K.B.: Recurrent neural network, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_7

Deep Learning for Character Recognition

B. R. Kavitha, Govindaraj Ramya, and P. Kumaresan

1 Character Recognition

Character recognition is a computer vision application which deals with how a machine interprets a text which is presented as an image. Machines understand text by their unicode formats. Every language has a set of characters which are identified by a computer by their unicode format. But what if the text is presented as an image? An image is represented by a collection of pixel values. All that a machine could get from the image is pixel values. We need algorithms to determine what is in the image, whether an animal or text or human face or car. Recognizing the text that is present in the image is one of the challenging computer vision tasks.

Text can be present in documents that are scanned or captured as photo or in natural images, name boards, number plates, postcards, and postal covers which may be printed, typed, or handwritten characters in Fig. 1.

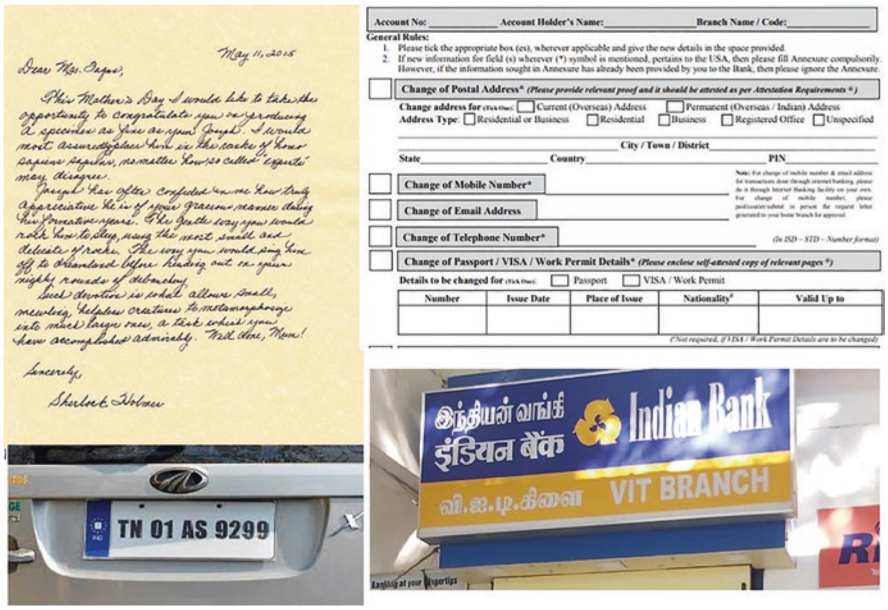
The conversion of text images to machine-encoded text is referred as optical character recognition (OCR). OCR is classified into two categories: online and offline OCRs. Online OCRs generally refer to online recognition of handwritten characters. The online handwriting would be acquired through the strokes written on a digital surface using a special pen. Offline character recognition refers to any printed, typed, or handwritten text which is captured as image either by scanning or taking a photo. The size of online handwritten data would be less when compared to offline since offline data consist of pixel intensities.

Scene text recognition refers to identification of text present in natural images. Firstly, the text in the scene should be localized, and then the script of the language of the text should be determined. It may be a single language or multi-lingual.

B. R. Kavitha (✉) · G. Ramya · P. Kumaresan
Vellore Institute of Technology, Vellore, TN, India
e-mail: kavitha.br@vit.ac.in; ramya.g@vit.ac.in; pkumaresan@vit.ac.in

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_3



May 11, 2015

Dear Mrs. [illegible],

This Mother's Day I would like to take the opportunity to congratulate you on producing a specimen as fine as your Joseph. I would most assuredly place him in the ranks of homo sapiens sapiens, no matter how so called "experts" may disagree.

Joseph has often confided in me how truly appreciative he is of your gracious manner during his formative years. The gentle way you would rock him to sleep, using the most small and delicate of rocks. The way you would sing him off to dreamland before heading out on your nightly rounds of debauchery.

Such devotion is what allows small, mewling, helpless creatures to metamorphosize into much larger ones, a task which you have accomplished admirably. Well done, Mom!

Sincerely,

Sherlock Holmes

Account No: ______ **Account Holder's Name:** ______ **Branch Name / Code:** ______

General Rules:

1. Please tick the appropriate box (es), wherever applicable and give the new details in the space provided.
2. If new information for field (s) wherever (*) symbol is mentioned, pertains to the USA, then please fill Annexure compulsorily. However, if the information sought in Annexure has already been provided by you to the Bank, then please ignore the Annexure.

☐ **Change of Postal Address*** ***(Please provide relevant proof and it should be attested as per Attestation Requirements *)***

Change address for (Tick One): ☐ Current (Overseas) Address ☐ Permanent (Overseas / Indian) Address

Address Type: ☐ Residential or Business ☐ Residential ☐ Business ☐ Registered Office ☐ Unspecified

______ **City / Town / District** ______

State ______ **Country** ______ **PIN** ______

☐ **Change of Mobile Number*** ______

☐ **Change of Email Address** ______

Note: For change of mobile number & email address for transactions done through internet banking, please do it through Internet Banking facility on your own. For change of mobile number, please post/courier/submit in person the request letter generated to your home branch for approval.

☐ **Change of Telephone Number*** ______ *(In ISD – STD – Number format)*

☐ **Change of Passport / VISA / Work Permit Details*** ***(Please enclose self-attested copy of relevant pages *)***

Details to be changed for (Tick One): ☐ Passport ☐ VISA / Work Permit

Number	Issue Date	Place of Issue	Nationality#	Valid Up to

(#Not required, if VISA / Work Permit Details are to be changed)

Fig. 1 Sample images containing text

English text and numerals are the mostly seen language in natural images. Secondly, the word or words should be segmented, or the individual characters should be segmented. In case of words, the complete word itself should be recognized using dictionary, and for characters, each character should be recognized. For character recognition, in case of natural images, it has to identify the text area, segment each character and recognize the characters, and present it in a machine understandable format. This chapter explains the recognition of isolated individual characters by implementing convolutional neural networks, one of the most prominent deep learning methods.

Applications Character recognition is an application which is widely used these days with the evolution of more and more gadgets. For example, Google Lens is a well-known application which captures the text as image and converts it into a machine-encoded format which can further be translated to any native language. As we are moving toward smart cities and more Internet-connected things, we need to automate the understanding of text in many scenarios at a faster pace. An automated vehicle should be able to read the name boards in addition to the sign boards and act accordingly. Or a person who does not know the native language of a particular place can just scan the text and translate it to a language which he/she understands easily.

1.1 Challenges in Character Recognition

The common challenges are variations in viewpoint, scale, deformations, light intensities, background clutter, and occlusion.

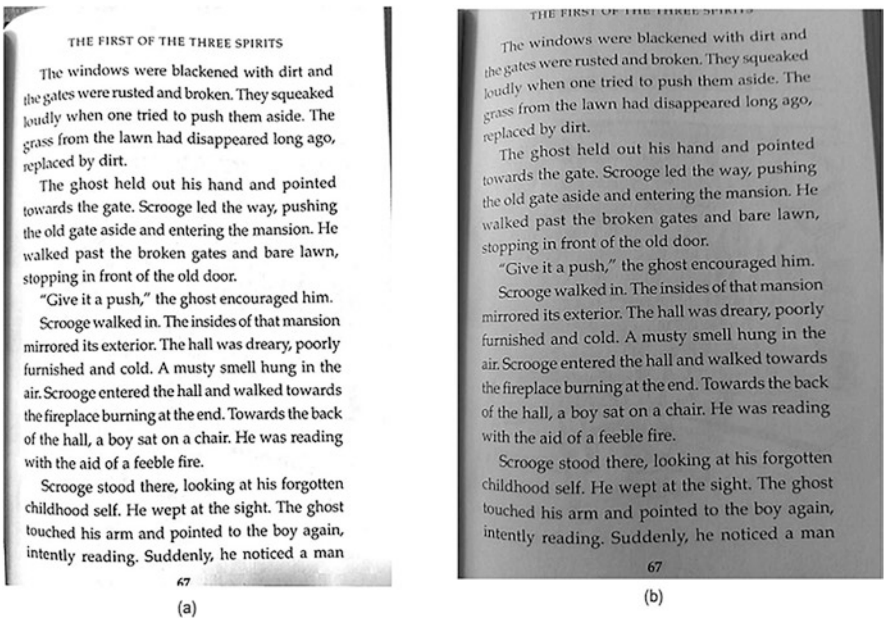
In case of document recognition, data acquisition could have happened at different conditions such as illuminance, captured devices, distance, angle, scale, etc.

The document images may be obtained from a high-quality scanner or from a mobile camera. The quality of both the images may vary as shown in Fig. 2. Earlier, to digitize a paper document, the document needs to be scanned by a scanner with high resolution. The present scenario is a document can be scanned using mobile devices such as smart phones or tablets. It can be directly taken as images or using some specialized mobile applications like Adobe scanner. The images obtained from high-quality scanners would have uniform lighting taken under a flat surface without any distortions.

The images from a smart phone or digital camera may vary in any of the following conditions:

View Point The text image could have been captured from different angles and from varied distances. The image may be zoomed in or taken from a longer shot.

Size The size of the text may be smaller or bigger in real which when captured as image would result in text of different scaling.


THE FIRST OF THE THREE SPIRITS

The windows were blackened with dirt and the gates were rusted and broken. They squeaked loudly when one tried to push them aside. The grass from the lawn had disappeared long ago, replaced by dirt.

The ghost held out his hand and pointed towards the gate. Scrooge led the way, pushing the old gate aside and entering the mansion. He walked past the broken gates and bare lawn, stopping in front of the old door.

"Give it a push," the ghost encouraged him.

Scrooge walked in. The insides of that mansion mirrored its exterior. The hall was dreary, poorly furnished and cold. A musty smell hung in the air. Scrooge entered the hall and walked towards the fireplace burning at the end. Towards the back of the hall, a boy sat on a chair. He was reading with the aid of a feeble fire.

Scrooge stood there, looking at his forgotten childhood self. He wept at the sight. The ghost touched his arm and pointed to the boy again, intently reading. Suddenly, he noticed a man

67

THE FIRST OF THE THREE SPIRITS

The windows were blackened with dirt and the gates were rusted and broken. They squeaked loudly when one tried to push them aside. The grass from the lawn had disappeared long ago, replaced by dirt.

The ghost held out his hand and pointed towards the gate. Scrooge led the way, pushing the old gate aside and entering the mansion. He walked past the broken gates and bare lawn, stopping in front of the old door.

"Give it a push," the ghost encouraged him.

Scrooge walked in. The insides of that mansion mirrored its exterior. The hall was dreary, poorly furnished and cold. A musty smell hung in the air. Scrooge entered the hall and walked towards the fireplace burning at the end. Towards the back of the hall, a boy sat on a chair. He was reading with the aid of a feeble fire.

Scrooge stood there, looking at his forgotten childhood self. He wept at the sight. The ghost touched his arm and pointed to the boy again, intently reading. Suddenly, he noticed a man

67

Fig. 2 (**a**) Scanned document (**b**) Document taken as photo

Warping In case of text documents, the paper may be crumpled leading to distorted image.

Cluttered Background The background of the document images may have clutter or taken in an inconsistent nonplanar background as compared to a document scanned from a scanner.

Lighting Conditions The external lighting from direct sunlight, flashlights, and other lighting sources affects the image quality to an extent. Uneven lighting or poor lighting will result in low-quality images where the text may not be clear.

Font Type The same text in two different images could have different font types. In case of handwritten text, there would be variations in the handwriting of every individual.

2 Deep Learning Approach on Character Recognition

Deep learning has proved its effect on many computer vision applications successfully with almost or in some cases even better than human accuracy. Lots of work have happened till last decade using traditional machine learning approaches. The core component of any machine learning is feature extraction which is done by many algorithms such as edge detector, SIFT, SURF, and ORB. This feature extraction is automated in deep learning method. Deep learning is commonly referred as simulation of the learning pattern of the human brain. Given a set of examples, the features are automatically inferred from them in deep learning.

Though other deep learning models work fine for some of the computer vision applications, convolutional neural networks work the best for many of them and character recognition as well. LeNet5 which was developed by Yann LeCun was the earliest CNN for recognition of handwritten digits [1, 2]. In 2012, a multi-column deep neural network (MCDNN) was developed by Ciregan et al. [3] and was tested on numbers, Latin characters, and Chinese characters. But CNNs became popular only after its breakthrough results in ImageNet competition in 2012 [4]. Researchers started looking at CNN as the best solution and tried every computer vision application with CNN.

2.1 Convolutional Neural Networks

CNNs comprise a stack of one or more of the following layers: convolution layer (conv layer), pooling layer (pool layer), and fully connected layer (fc layer). Every conv layer would be passed with a set of filters, i.e., the features, which are searched in every training data and saved as feature maps. The initial layers would look for simple geometric shapes such as lines, curves, and edges. The consecutive layers would look for complex components such as circles and squares.

Pooling layer would decrease the number of computations by downsizing the number of pixels. This would take the summary of a part of the region which determines whether a feature is present in that region or not. In max pooling technique, the pixel which has a maximum value in the neighborhood of the n x n window is taken, whereas in average pooling technique, the average of all the pixels in the neighborhood of *n x n* window is taken.

FC layers are similar to hidden layers of normal neural networks which flattens the region-based values obtained from convolution and pooling layers as a single vector. These layers would look for the combinations of the features identified in initial layers. Suppose in English language two perpendicular lines represent the letter L. But T also has two perpendicular lines. Here, the position or location of these two lines would also help us to determine the character which is also a feature. When these features are present in an image, it is recognized as L or T.

A typical workflow of a character recognition system using a CNN model is shown in Fig. 3.

How Do CNNs Differ from Traditional Networks? A classic neural network would connect each of the neurons of one layer to each of the neurons in the following layer. As a result of this, more number of parameters would be required to train a normal neural network. In case of CNN, the spatial information and the relation between the neighboring pixels in a small region are maintained. In normal neural networks, every pixel is an input to the nodes in input layer. Hence, the spatial information is lost.

Activation Function In neural networks, neurons get activated if the input value is above a threshold value. The most widely adopted activation function since the CNN breakthrough is rectified linear units (ReLU) since it is simpler in computation yet yields faster convergence. This function takes the form z = (0,max) which would result in values less than zero becoming zero and all other values remaining

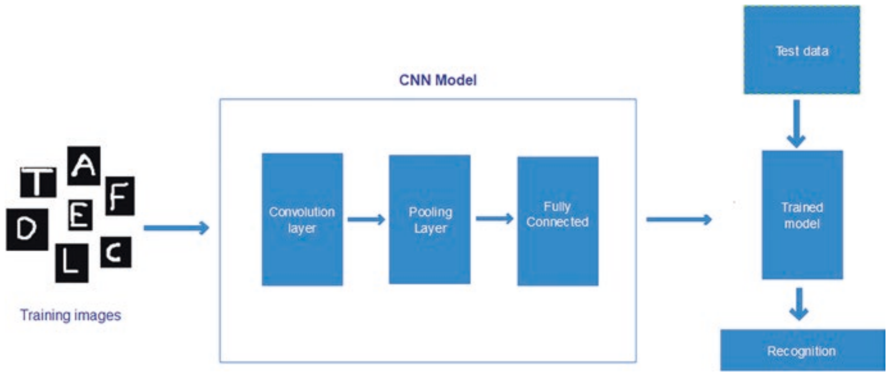


Fig. 3 A character recognition workflow using CNN

the same. Several types of ReLU have evolved such as Leaky ReLU (LReLU), Parameterized ReLU (PReLU), and Exponential linear unit (eLU).

Loss Function Loss functions determine the error while updating the weights. During training process, for every forward propagation, the weights are calculated, and the difference with the actual value is obtained. This error is minimized over a number of iterations by updating the weights during backward propagation.

Optimizer Optimization algorithms are required to minimize the loss function. While back propagating the network, these algorithms compute the gradients that would help in reaching the minimum cost function. Various optimization algorithms are available such as stochastic gradient descent (SGD), RMSprop, Adam, Adagrad, Adadelta, and many more. All the optimizers have other parameters such as learning rate and momentum which can be fine-tuned based upon the application for better results.

Batch Size Instead of computing the gradients for all the training data for every iteration which would require more memory for larger datasets, the data samples can be passed as batches, and the updation of parameters can happen for every batch. Based on the memory capacity the batch size can be fixed.

3 Review on Various Character Sets

There are many languages that originated from different parts of the world, and some of them had spread across the world. These languages are written using various scripts. Most of the languages have their own script, while some scripts are shared by multiple languages. All these characters have their unique unicode format understandable by computers.

Numerals The numerals representing 0–9 are used to represent the numbers in Latin script derived from Hindu-Arabic numeral system which is the most widely used script for numbers across the world. The standard dataset used for numbers is Modified National Institute of Standards and Technology (MNIST) dataset [5]. This dataset is a collection of digit images (0–9) that were handwritten and is basically derived from a larger dataset, NIST database. The data was collected from a group of 250 members of different ages comprising of students and employees of Census Bureau organization. It comprises of 70,000 gray scale handwritten digit images (split as 60,000 images for training and 10,000 images for testing) of 28×28 dimensions which have the digit centered in 20×20. This dataset had been the most tested dataset by many machine learning algorithms for computer vision applications. For researchers or developers of deep learning models, this dataset would be the ideal choice of their first test dataset (something like “Helloworld” of machine learning algorithms).

English Characters Latin script is used to write English characters. The 26 uppercase and lowercase alphabets contained in the Latin script were standardized by International Standards Organization (ISO) in the 1960s which was earlier standardized by the American Standard Code for Information Interchange (ASCII).

NIST Special Database 19 dataset [6] contains the collection of handwritten lowercase letters (a-z), uppercase letters (A-Z), and digits (0–9). A total of 7,31,668 training data and 82,587 test data is available in this database. This dataset's structure is slightly complicated, and hence the access to this dataset is not easier. A variant of NIST database is now available as Extended MNIST (EMNIST) [7] which has the similar structure of MNIST dataset.

Chinese Characters Chinese characters are used to write the Chinese language. It is also used to write many other Asian languages such as Japanese (known as kanji), Korean, and Vietnamese. Chinese characters are vast in number more than ten thousand, but the most commonly used Chinese characters range from 3000 to 4000. They have evolved from traditional to simplified Chinese characters with reduced number of strokes. The Institute of Automation of the Chinese Academy of Sciences (CASIA) and National Laboratory of Pattern Recognition (NLPR) have developed datasets OLHWDB for online handwritten database and HWDB for offline handwritten database which consists of nearly one million character samples of 3755 classes with 300 samples per class [8]. Chinese character recognition has shown best results in spite of their highest number of characters. Some of their works are reported in [9–11].

Arabic Characters Very few scripts have the method of writing from right to left. One such script is Arabic script which consists of 28 characters in the first level, and each character can take different shapes based on its position in the word. OIHACDB and AHCD are some benchmarked datasets of Arabic characters. OIHACDB has 40 classes with 30,000 images, and AHCD has 16,800 images for 28 classes [12]. Arabic character recognition also has shown best results with deep learning methodologies [13, 14].

Bangla Characters Languages such as Bengali, Assamese, and Manipuri use Bangla script as their script. It basically consists of 50 characters and 10 numerals. Though there are several compound characters, not all of them are present in Bangla datasets. Several standard datasets are available for Bangla scripts [15–18]. In Indic scripts, for Bangla character recognition, various deep learning methodologies were explored [19, 20].

Scene Text Dataset Detection and recognition of text from natural images is a challenging task compared to scanned documents because of its non-uniform background. Several databases are available for scene text detection and individual character recognition. Robust Reading Competition conducted by ICDAR mainly focuses on detecting and recognizing the text on natural images [21]. Chars74K dataset [22] is a collection of 74,000 natural images which has English alphabets

Table 1 Some of the character datasets of various scripts

Script (language)	Dataset	No. of classes	No. of training samples
Numbers	MNIST	10	60,000
Latin (English)	NIST SD-19	62	7,31,668
Chinese	CASIA (OLHWDB1.1)	3755	11,23,132
Tamil	HP Labs	156	82,928
Arabic	AHCD	28	16,800
	OIHACDB	40	30,000
Bangla	ISIHCD	50	37,858

uppercase (A-Z), lowercase (a-z), numbers (0–9), and Kannada characters. COCO-Text dataset [23] consists of 63,000 images of printed as well as handwritten text having natural image background with over 173k text annotations. Chinese natural scene text detection was presented in [24].

Various character datasets along with their number of training samples and number of classes are displayed in Table 1.

4 Implementation of Character Recognition Using Keras and TensorFlow

TensorFlow TensorFlow is a Python-based open-source platform developed by Google Brain team for implementing machine learning algorithms [25]. It can be used on a wide range of heterogeneous systems starting from a large number of distributed systems to mobile devices and embedded GPU devices. It can run on various computing platforms such as Linux, Windows, macOS, Android, and iOS.

TensorFlow is derived from two keywords: “tensor” which refers to multidimensional data representation and “flow” which refers to the dataflow programming model.

Keras Keras is an easy-to-use machine learning API developed mainly to build deep learning models using Python. It functions over the deep learning frameworks such as TensorFlow and Theano which use tensors, a multidimensional array for its computations. Keras with TensorFlow as backend is the widely used framework. This would be the right choice for beginners since it provides abstraction, is easy to learn, and has faster implementation. Without the knowledge of the underlying mathematical computations, the APIs could be used for building a model, and training can be done.

Model Model is the basic building block of Keras. A model can be defined, compiled, and fitted in order to predict new data. A neural network is represented by combining at least three layers, input, output, and hidden layers. Deep CNNs consist of many hidden layers. Convolutional neural networks have a special kind of layers such as convolutional layer, pooling layer, and activation layer. Keras models are used to construct these neural network models by concatenating these layers.

Two ways of building these models are sequential model and functional API models. Sequential models are constructed by stacking layers one over another, while functional API takes the input to the layer and output from the layer as two parameters for defining each layer. Sequential models are simple and easier to construct and provide a high level of abstraction which is mainly used for building simple models. Keras has many built-in pretrained models like VGG, ResNet, and Inception which can be used to train on new data.

Functional APIs are used for building complex models. They can be used to work on multi-inputs and multi-outputs or multi-inputs and single output. These APIs provide more flexibility to build models with shared layers such as in ResNet. Several methods are available in Keras for employing various functions of machine learning algorithms such as compile, train, fit, evaluate, and predict.

Steps for Developing a CNN Model for Character Recognition This section explains how to build a convolutional neural network for recognizing handwritten Tamil characters. The dataset for Tamil characters is available as Isolated Handwritten Tamil Character Dataset by HP Labs, India [26]. It is composed of nearly 82,000 training images comprising of 156 classes. For our example, we would take only 13 classes of this Tamil character dataset. Some example images of these 13 classes are displayed in Fig. 4. This subset contains approximately 500 sample images for every class in train dataset and exactly 50 sample images for every class in test set. The images are of different sizes hence resized to a fixed size of (64,64). These resized images are recasted to binary form, inverted, and converted to a numpy array.

The following steps are followed for developing a CNN model:

- Create model
- Compile model
- Train model
- Test model

Create Model Here, we will create a sequential model of a basic convolutional neural network.

Importing Libraries
Firstly, we will import the Keras libraries that are needed for the construction, compilation, and training of a model.

```
 from tensorflow.keras import models, layers, losses
 from tensorflow.keras.layers import Conv2D,MaxPooling2D
   from   tensorflow.keras.layers   import   Dense,   Flatten,
Activation, Dropout
 from tensorflow.keras.optimizers import Adam
```

For creating a sequential model, the following method is used.

```
model=models.Sequential()
```

Adding Layers

Layers can be added using *add()* function. The type of the layer to be added is passed as a parameter to the function.

To add a convolutional layer, the following method is used,

```
model.add(Conv2D(nof, (fw, fh), padding="same", name="Conv1",
input_shape=(width,height,channels)))
```

where *Conv2D* is a function which represents two-dimensional convolutional layer and takes the following parameters:

nof – Number of filters.
(fw, fh) – Filter size as comma-separated values (width, height).
Padding – No zero padding, same-zero padding.
name – Name of the convolutional layer.
input_shape – For the first layer alone, the shape of the input is passed as another parameter. Shape of the input dimensions takes the format (width, height, channels).

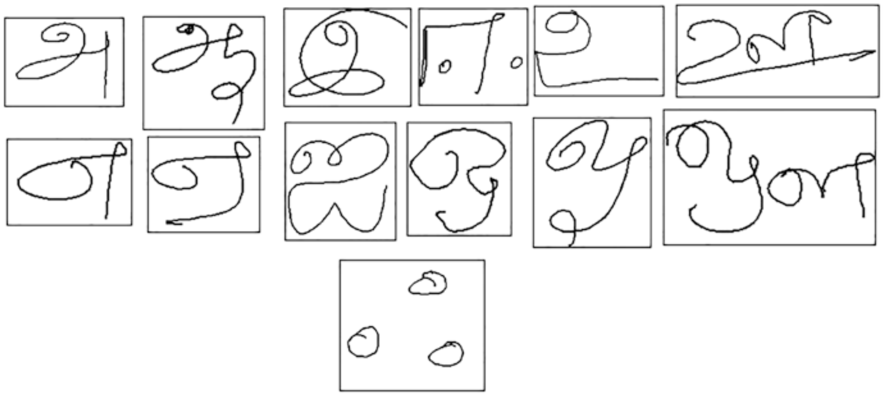


Fig. 4 A subset of 13 characters of Tamil character dataset

To add an activation layer,

where *Activation* function represents the activation function to be employed on the

```
model.add(Activation("relu"))
```

feature maps. The name of the activation function is passed as parameter. Some of the activation function values are relu, elu, lrelu, prelu, tanh, and many more.

To add a pooling layer, *add()* function is passed with a pooling function like,

where *MaxPooling2D* function represents max pooling function with the pooling

```
model.add(MaxPooling2D(pool_size=(2,2)))
```

window size as parameter. Other pooling functions such as *AveragePooling2D* and *GlobalAveragePooling* can also be used.

For any convolution neural network, these three layers would make the feature extraction part. Combinations of these three layers are stacked one over another. The number of layers is also a parameter in defining the model, and any number of such combinations can be made. In the example, there are three such combinations of layers.

The output of the last convolution layer is connected to one or more fully connected layers. Before connecting to a fc layer, the output from conv layer must be converted to a one-dimensional vector using *Flatten()* function. This function should be passed to *add()* function without any parameter.

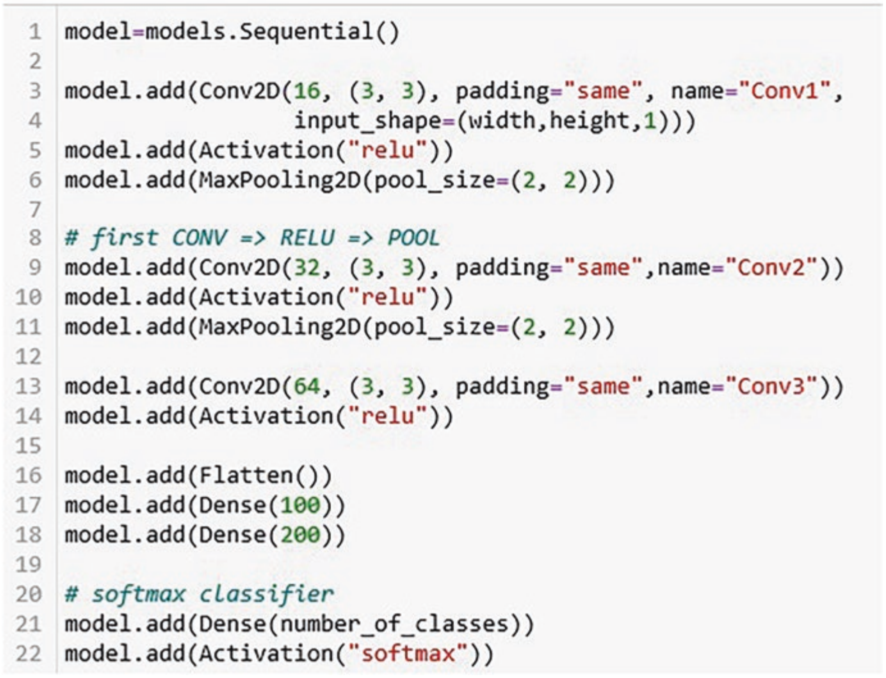
To add a fc layer, *Dense()* function is used. The flatten function and Dense function are added by,

The final layer is the output layer which has *n* number of neurons where n corresponds to the number of classes. It is also added by Dense function, and it requires

```
model.add(Flatten())
model.add(Dense(No.of.neurons))
```

an activation function, which is generally a softmax function in case of multiclass classifier or sigmoid in case of binary classification.

```
model.add(Dense(number_of_classes))
model.add(Activation(classifier))
```



```
model=models.Sequential()

model.add(Conv2D(16, (3, 3), padding="same", name="Conv1",
                    input_shape=(width,height,1)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

# first CONV => RELU => POOL
model.add(Conv2D(32, (3, 3), padding="same",name="Conv2"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding="same",name="Conv3"))
model.add(Activation("relu"))

model.add(Flatten())
model.add(Dense(100))
model.add(Dense(200))

# softmax classifier
model.add(Dense(number_of_classes))
model.add(Activation("softmax"))
```

Fig. 5 Code for defining a model for character recognition

The above steps are composed together and displayed in Fig. 5.

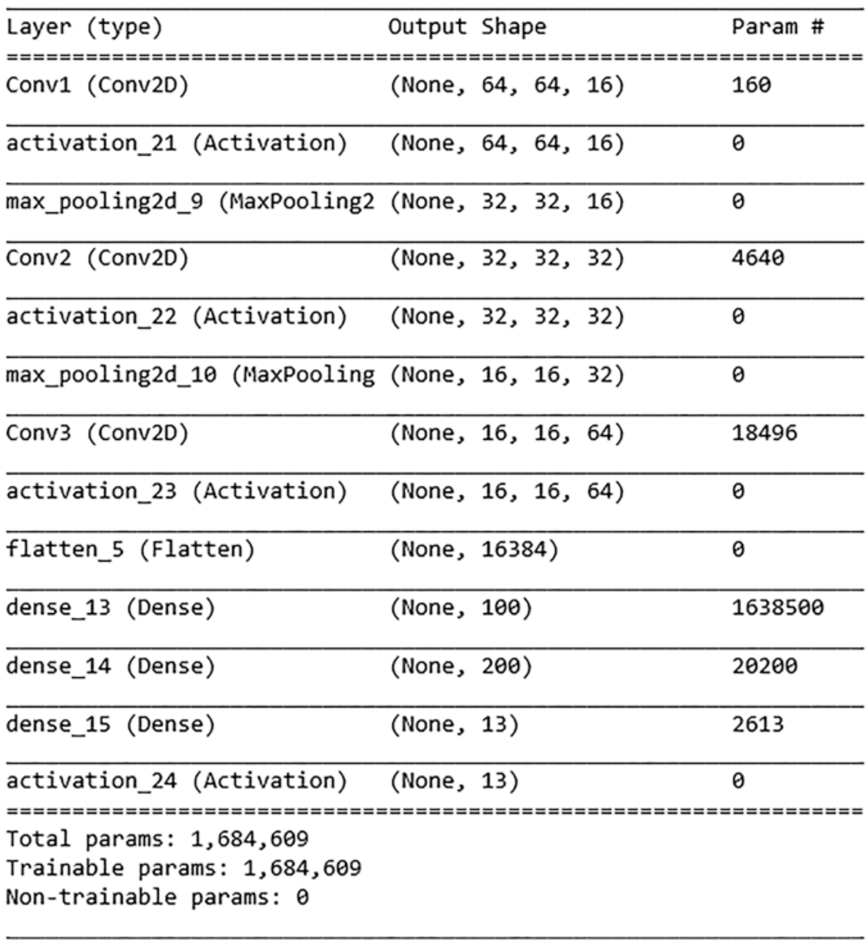
The model is created, and the overall architecture of the model can be viewed by *model.summary()* as displayed in Fig. 6. This would give the detailed information about the total number of parameters used for training and layer-wise input and output parameter details.

Compile Model Now the model has been defined. This model should be compiled with specific values for parameters and trained with input set data. Keras has *model.compile()* function to perform this. It specifies the choice of optimizer and loss function. The following parameters are passed to this function.

```
model.compile(optimizer, loss, metrics)
```

optimizer – Optimizer values can be sgd, rmsprop, Adam, Ada, and Adagrad. The parameters of the optimizers such as learning rate and momentum can also be set.

Loss – Loss functions can be any of the following: mean squared error, mean absolute error, categorical crossentropy, and binary crossentropy.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Conv1 (Conv2D)               (None, 64, 64, 16)        160
_________________________________________________________________
activation_21 (Activation)   (None, 64, 64, 16)        0
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 32, 32, 16)        0
_________________________________________________________________
Conv2 (Conv2D)               (None, 32, 32, 32)        4640
_________________________________________________________________
activation_22 (Activation)   (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 16, 16, 32)        0
_________________________________________________________________
Conv3 (Conv2D)               (None, 16, 16, 64)        18496
_________________________________________________________________
activation_23 (Activation)   (None, 16, 16, 64)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_13 (Dense)             (None, 100)               1638500
_________________________________________________________________
dense_14 (Dense)             (None, 200)               20200
_________________________________________________________________
dense_15 (Dense)             (None, 13)                2613
_________________________________________________________________
activation_24 (Activation)   (None, 13)                0
=================================================================
Total params: 1,684,609
Trainable params: 1,684,609
Non-trainable params: 0
_________________________________________________________________
```

Fig. 6 Summary of the model

Metrics – This parameter is used for the evaluation of the model performance. Some o f the values can be accuracy, binary_accuracy, and top_k_categorical_accuracy.

An example of setting the optimizer, loss function, and metrics is given below:

```
optimizer=Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0)
loss="categorical_crossentropy"
model.compile(loss=loss,optimizer= optimizer,
metrics=['accuracy'])
```

Train Model Training the model is the learning part of the machine which would enable the machine to recognize a new character.

The segmented, cropped, isolated character images are fed to the model. These images form the training dataset. The training dataset would be used by the model to learn the features extracted from the character images. Updation of weights can happen based on the training dataset. The volume, variety, and quality of training data would result in a model with high accuracy.

In addition to the training set, there exists another dataset known as validation set which is split from training data. This validation dataset is required during training to evaluate the model's performance. After one complete traversal of the entire training set in both the directions (forward and backward) over the model, the validation dataset is fed to the model, and accuracy is determined. The comparison between the training accuracy and validation accuracy would help in understanding the learning of the model. If the validation accuracy is far less than training accuracy, it is understood that the model is overfitting, which proves that the model is more specific to the training data. If the training accuracy and validation accuracy both are very less, then the model is underfitting such that the model did not learn any good features.

In Keras, the splitting of training set and validation set can be done beforehand or during runtime. The following code splits the training and validation set during runtime.

```
X_trainset, X_testset, y_trainlbl, y_testlbl
=train_test_split(x, y,
test_size=0.2, random_state=2)
```

This *test* represents the validation set and not test set. (Note: Test data will be discussed in next subsection.) Values *x* and *y*l denote the training data and their labels respectively. The splitting of training data and validation data can be of any ratios such as 90:10, 80:20, or 60:40.

In the example, we have made the split as 80:20 by specifying the *test_size* parameter as 0.2. The function returns training images(X_trainset), validation images(X_testset), training labels(y_trainlbl), and validation labels(y_testlbl).

The model is trained using *fit()* method for a certain number of epochs with the complete training data. The validation data is tested after every epoch. Both the accuracies and losses are recorded.

```
hist=model.fit(X_trainset,y_trainlbl, batch_size=64,
epochs=30, verbose=1,
validation_data=(X_testset, y_testlbl))
```

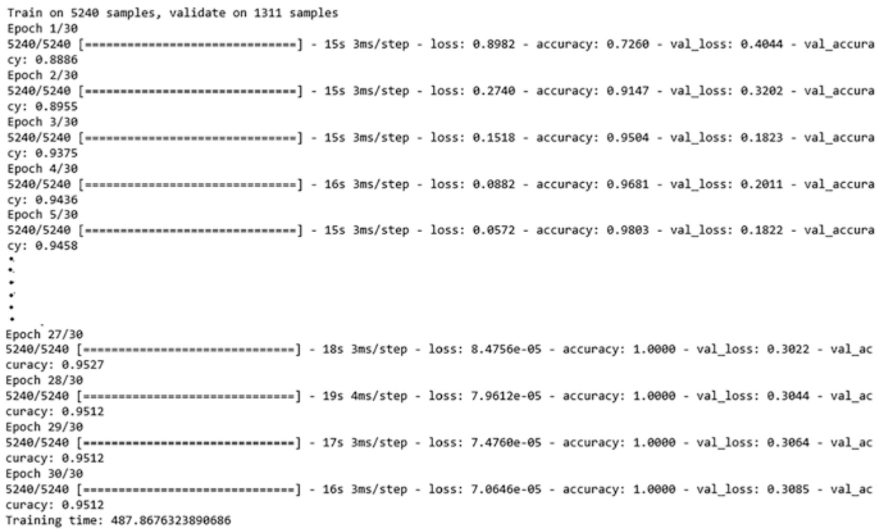
The following parameters are passed as arguments:

X_trainset – Training images as numpy array
y_trainlbl – Training labels
batch_size – Number of training images fed to the model at a time
epochs – Number of epochs to train the model
verbose – Verbosity which can have values (0,1,2)
validation_data – Validation images as numpy array(X_testset), labels(y_testlbl)}

The training process is run with verbose 1 for 30 epochs which is displayed in Fig. 7. The first line in the screenshot shows the training and validation split of the samples. Then, for every epoch, the accuracy and loss measured for training and validation and accuracy are reported. It shows the time taken to train for every epoch. The training accuracy and validation accuracy of this model are 100% and 95.12%, respectively. This means that the model is overfitting, and it is needed to add some regularization methods.

It returns a history object which gives the details of losses and accuracies at every epoch. The trained model can be saved and used for testing new data at any time. In Keras, the weights and model can be saved in .h5 extension.

Test Model Once the model is trained with known training images with better accuracy, it can be tested with unknown new test data. Ideally the testing dataset should not be from the same cohort of train dataset. Testing is done using *predict()* method which predicts the output for the test images. The predicted output labels are returned as numpy arrays.



```
Train on 5240 samples, validate on 1311 samples
Epoch 1/30
5240/5240 [==============================] - 15s 3ms/step - loss: 0.8982 - accuracy: 0.7260 - val_loss: 0.4044 - val_accura
cy: 0.8886
Epoch 2/30
5240/5240 [==============================] - 15s 3ms/step - loss: 0.2740 - accuracy: 0.9147 - val_loss: 0.3202 - val_accura
cy: 0.8955
Epoch 3/30
5240/5240 [==============================] - 15s 3ms/step - loss: 0.1518 - accuracy: 0.9504 - val_loss: 0.1823 - val_accura
cy: 0.9375
Epoch 4/30
5240/5240 [==============================] - 16s 3ms/step - loss: 0.0882 - accuracy: 0.9681 - val_loss: 0.2011 - val_accura
cy: 0.9436
Epoch 5/30
5240/5240 [==============================] - 15s 3ms/step - loss: 0.0572 - accuracy: 0.9803 - val_loss: 0.1822 - val_accura
cy: 0.9458
.
.
.
.
.
.
Epoch 27/30
5240/5240 [==============================] - 18s 3ms/step - loss: 8.4756e-05 - accuracy: 1.0000 - val_loss: 0.3022 - val_ac
curacy: 0.9527
Epoch 28/30
5240/5240 [==============================] - 19s 4ms/step - loss: 7.9612e-05 - accuracy: 1.0000 - val_loss: 0.3044 - val_ac
curacy: 0.9512
Epoch 29/30
5240/5240 [==============================] - 17s 3ms/step - loss: 7.4760e-05 - accuracy: 1.0000 - val_loss: 0.3064 - val_ac
curacy: 0.9512
Epoch 30/30
5240/5240 [==============================] - 16s 3ms/step - loss: 7.0646e-05 - accuracy: 1.0000 - val_loss: 0.3085 - val_ac
curacy: 0.9512
Training time: 487.8676323890686
```

Fig. 7 Training process for Tamil character subset for 30 epochs

The test data should take the same format as training data (the preprocessing steps of training data should be applied for test data as well) and should be input with the same batch size as training data.

In order determine the accuracy of the test data, the predicted labels and actual

```
predictions = model.predict(test_data,batch_size=64)
```

labels are compared. The number of correctly predicted samples to the total number of predictions yields the accuracy of the test images. The code to determine the accuracy of the testing dataset is shown in Fig. 8. The *test_data* is a numpy array of preprocessed test images.

Tuning of CNN Model A CNN model can be modified to improve the performance by varying some of the parameters. The parameters can be the number of convolutional layers, pooling methods, choice of activation functions, optimizers or initializers, increasing/decreasing the batch sizes or number of epochs, training and validation split ratios, etc. These are first-level hyperparameters. The second level of hyperparameters could be the learning rate or momentum of optimizers. The above model was tried with different combinations of convolutional and pooling layers and has arrived at this model. The count of convolutional layers was increased to 5, max pooling layers to 3, and fully connected layers to 2. The number of filters is changed in all the convolutional layers. With the above changes, the number of parameters required for training has decreased from 1,684,609 to 875,677 parameters.

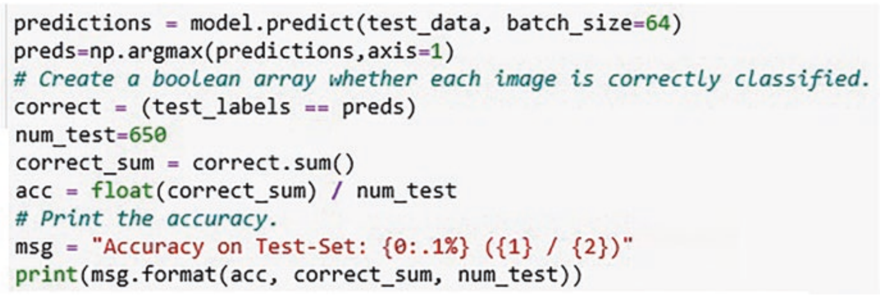
In addition to that, to avoid the overfitting of the model a popular technique called Dropout [27] is used, which drops some of the neurons to be inactive, thereby allowing the network to traverse through different architectures for every epoch. In Keras, we can specify the probability of number of neurons to be retained as parameter. The complete model with all the tuned parameters and dropout is shown in Fig. 9.

This model was compiled with *Adam* optimizer, and the learning rate was fixed as 0.001. The model was trained for 100 epochs, and the training time for 100 epochs is 47 minutes in a CPU machine (Fig. 10). With overfitting the validation accuracy has increased to 97.33% with a small decrease in training accuracy. The training, validation loss and training, validation accuracy are shown in Fig. 11 and Fig. 12 respectively. The accuracy on test set was reported as 95.2%.

The code for plotting the graphs is shown in Fig. 13. Using the history object returned from *model.fit()* function, the loss and accuracy are plotted. For plotting graphs, we need to import *matplotlib.pyplot* library.

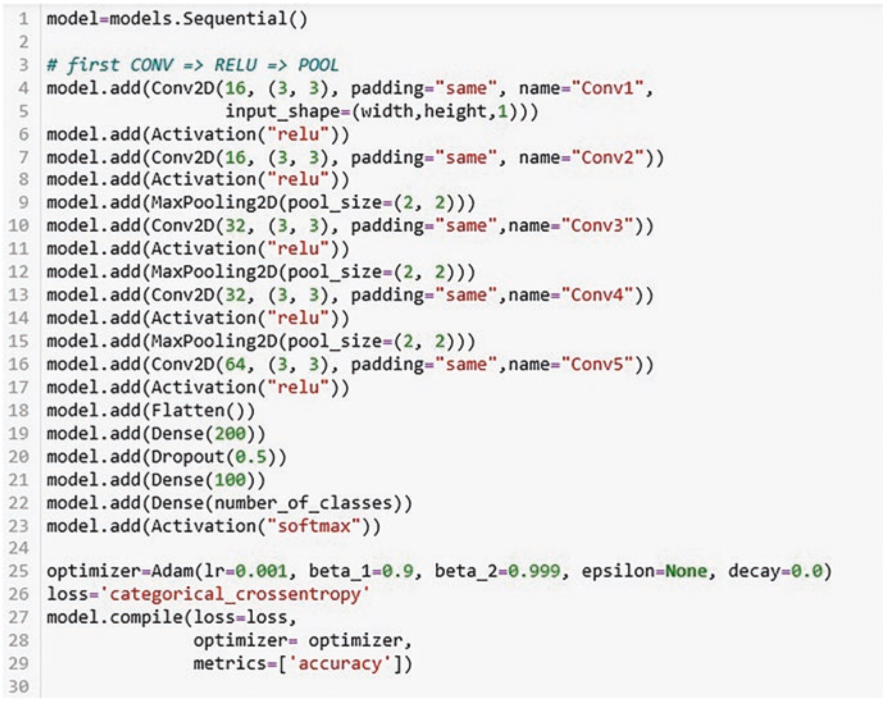
```
import matplotlib.pyplot as plt
```

Another metric that was used to measure the performance of the model was confusion matrix which is shown in Fig. 14. In the confusion matrix, true labels are in y axis, and predicted labels are in x axis. The diagonal specifies how many characters are classified correctly. For Class 0(Aa), Class 2(Ee), Class 4(Uu), and Class 12(Ak), all the characters are predicted correctly. Few characters are predicted wrongly because of similarity between characters and writing ambiguity. The highest number of misclassification is for Class 9(O) and Class 10(Oo) which is 5.



```python
predictions = model.predict(test_data, batch_size=64)
preds=np.argmax(predictions,axis=1)
# Create a boolean array whether each image is correctly classified.
correct = (test_labels == preds)
num_test=650
correct_sum = correct.sum()
acc = float(correct_sum) / num_test
# Print the accuracy.
msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
print(msg.format(acc, correct_sum, num_test))
```

Fig. 8 Code for recognition of test data



```python
model=models.Sequential()

# first CONV => RELU => POOL
model.add(Conv2D(16, (3, 3), padding="same", name="Conv1",
                 input_shape=(width,height,1)))
model.add(Activation("relu"))
model.add(Conv2D(16, (3, 3), padding="same", name="Conv2"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), padding="same",name="Conv3"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), padding="same",name="Conv4"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), padding="same",name="Conv5"))
model.add(Activation("relu"))
model.add(Flatten())
model.add(Dense(200))
model.add(Dropout(0.5))
model.add(Dense(100))
model.add(Dense(number_of_classes))
model.add(Activation("softmax"))

optimizer=Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0)
loss='categorical_crossentropy'
model.compile(loss=loss,
              optimizer= optimizer,
              metrics=['accuracy'])

```

Fig. 9 Code for tuned CNN model with dropout

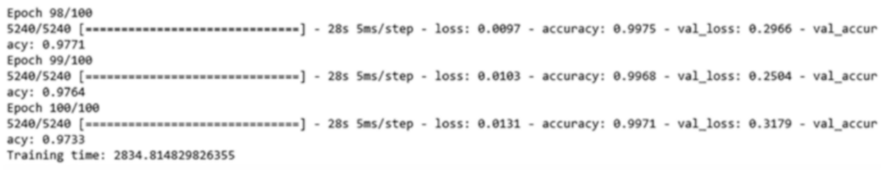

```
Epoch 98/100
5240/5240 [==============================] - 28s 5ms/step - loss: 0.0097 - accuracy: 0.9975 - val_loss: 0.2966 - val_accur
acy: 0.9771
Epoch 99/100
5240/5240 [==============================] - 28s 5ms/step - loss: 0.0103 - accuracy: 0.9968 - val_loss: 0.2504 - val_accur
acy: 0.9764
Epoch 100/100
5240/5240 [==============================] - 28s 5ms/step - loss: 0.0131 - accuracy: 0.9971 - val_loss: 0.3179 - val_accur
acy: 0.9733
Training time: 2834.814829826355
```

Fig. 10 Training of CNN model with dropout for 100 epochs

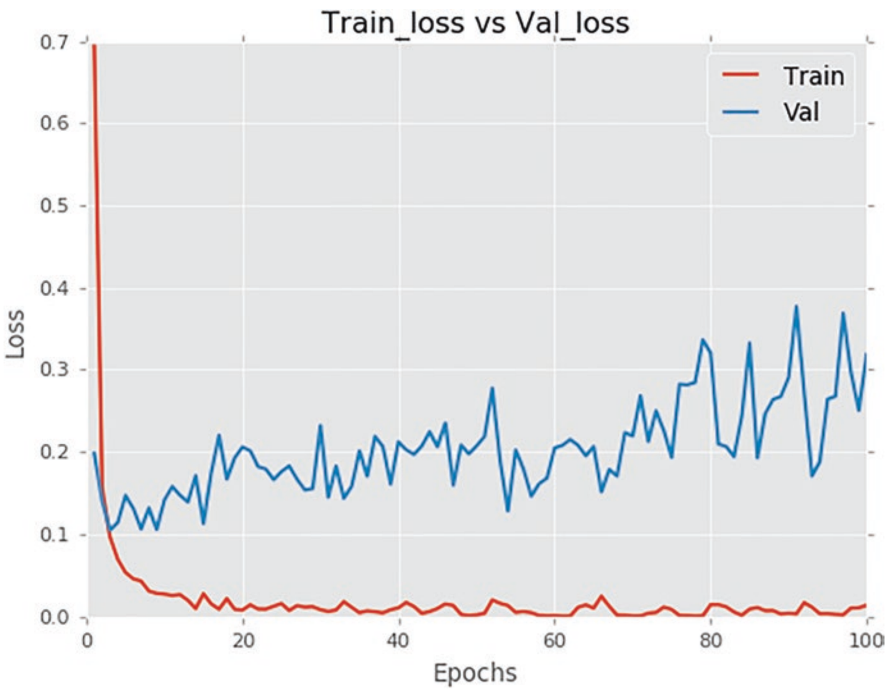


Fig. 11 Training loss and validation loss

Preprocessing In character recognition, the performance of the model during training may not be the same when a completely new character is tested. In case of handwritten characters, it will be even worse as the handwriting of every individual is different. It may not be possible to acquire all types of handwritten data. The training data could have been obtained from a cohort which belongs to a particular place or age. The data may not have larger variations. A model can exhibit the highest generalization only when the model is trained from a variety of data (all possible versions of the handwriting).

Hence it is better to modify the training data to a common representation before starting training. A lot of preprocessing is needed which would be followed by the new data as well so that it looks almost the training data.

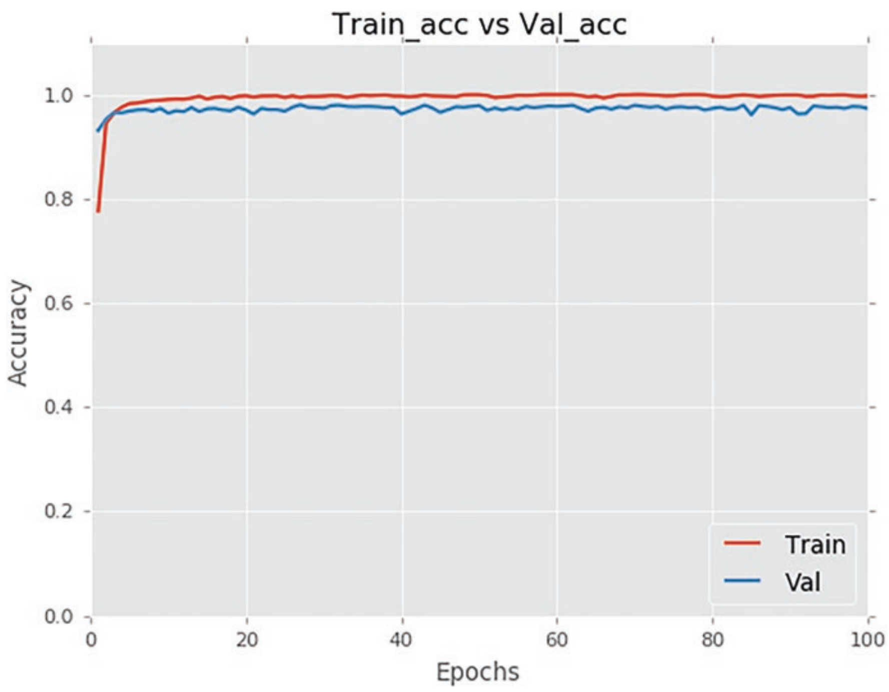


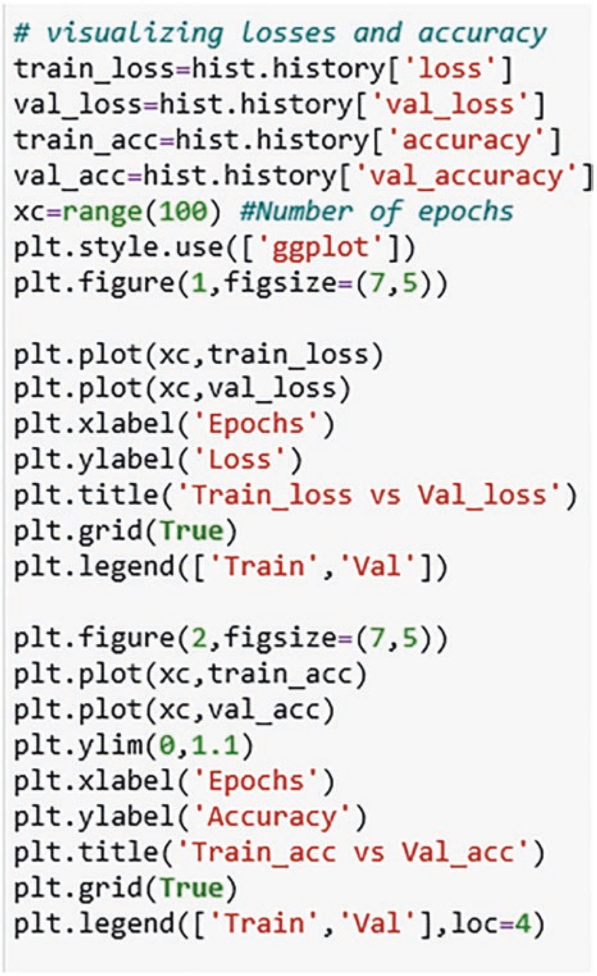
Fig. 12 Training and validation accuracy

In most of the cases, the characters that are segmented from the text image are converted to gray scale or binary image as the color of the text does not provide any useful information in most of the cases. Also, processing of binary or gray scale images is easier when compared to RGB images.

Some of the preprocessing methods used before training are discussed below. Practically, most of the preprocessing is done using opencv functions.

Size Normalization
The entire training images may not be of the same dimensions. Hence, all the images should be size normalized to a fixed dimension without losing the aspect ratio using any suitable interpolation method. Several interpolation methods are available, namely, bilinear interpolation, bicubic interpolation, etc.

Thresholding
This helps in separating the foreground image or object of interest from the remaining part of the image. This is achieved by converting all the pixel values above a threshold value to certain value and the remaining pixels to another value. Another method is using adaptive thresholding method which takes the neighbor pixel's value also into consideration so that different parts of the image can be tested with different threshold values [28–29].

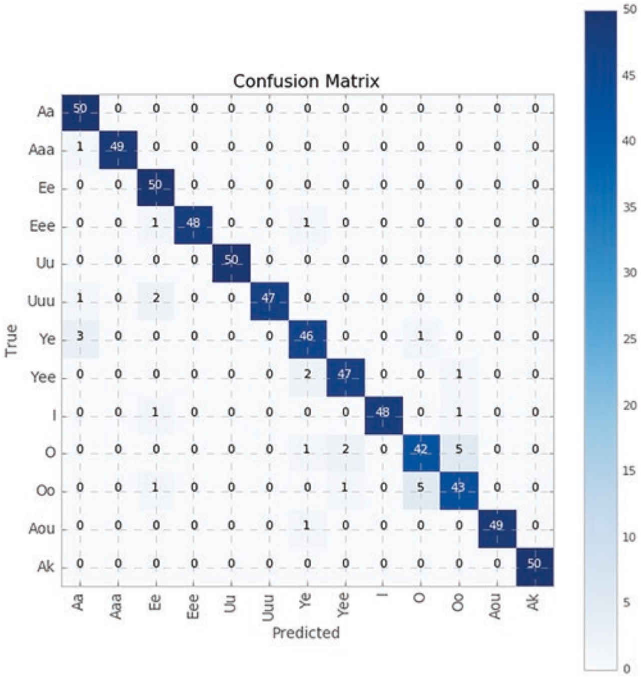


```
# visualizing losses and accuracy
train_loss=hist.history['loss']
val_loss=hist.history['val_loss']
train_acc=hist.history['accuracy']
val_acc=hist.history['val_accuracy']
xc=range(100) #Number of epochs
plt.style.use(['ggplot'])
plt.figure(1,figsize=(7,5))

plt.plot(xc,train_loss)
plt.plot(xc,val_loss)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train_loss vs Val_loss')
plt.grid(True)
plt.legend(['Train','Val'])

plt.figure(2,figsize=(7,5))
plt.plot(xc,train_acc)
plt.plot(xc,val_acc)
plt.ylim(0,1.1)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Train_acc vs Val_acc')
plt.grid(True)
plt.legend(['Train','Val'],loc=4)
```

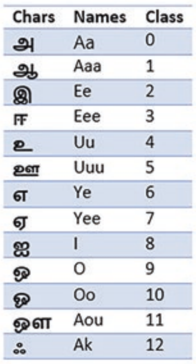
Fig. 13 The code for plotting the graphs

Dilation and Erosion
These are useful in improving the clarity of the character images which are broken or very thin. Dilation would increase the thickness of the character that is in foreground by assigning the value of a pixel to all the adjacent pixels of a fixed window size if the value is equal to 1. The reverse process of dilation is erosion which reduces the thickness of the character in foreground by discarding the pixels in neighborhood.

Centering
Whenever the text is segmented from a full document or from a scene text image, the spacing between characters may be very less. Hence, this process adds some blank borders on all the sides to center the character and ensure no information is lost.



True \ Predicted	Aa	Aaa	Ee	Eee	Uu	Uuu	Ye	Yee	I	O	Oo	Aou	Ak
Aa	50	0	0	0	0	0	0	0	0	0	0	0	0
Aaa	1	49	0	0	0	0	0	0	0	0	0	0	0
Ee	0	0	50	0	0	0	0	0	0	0	0	0	0
Eee	0	0	1	48	0	0	1	0	0	0	0	0	0
Uu	0	0	0	0	50	0	0	0	0	0	0	0	0
Uuu	1	0	2	0	0	47	0	0	0	0	0	0	0
Ye	3	0	0	0	0	0	46	0	0	1	0	0	0
Yee	0	0	0	0	0	0	2	47	0	0	1	0	0
I	0	0	1	0	0	0	0	0	48	0	1	0	0
O	0	0	0	0	0	0	1	2	0	42	5	0	0
Oo	0	0	1	0	0	0	0	1	0	5	43	0	0
Aou	0	0	0	0	0	0	1	0	0	0	0	49	0
Ak	0	0	0	0	0	0	0	0	0	0	0	0	50



Chars	Names	Class
அ	Aa	0
ஆ	Aaa	1
இ	Ee	2
ஈ	Eee	3
உ	Uu	4
ஊ	Uuu	5
எ	Ye	6
ஏ	Yee	7
ஐ	I	8
ஒ	O	9
ஓ	Oo	10
ஔ	Aou	11
ஃ	Ak	12

Fig. 14 Confusion matrix of the test data for 13 Tamil characters

Binarization
The process is converting the gray scale images to binary images, background with values 0 and foreground with values 1.

Scaling
Scaling is another normalization method which brings down all the samples in the same scale. Images will have varying pixel values based on the content. This process would scale down all the values between (0,1).

5 Summary

The notion of this chapter was to provide the basics of character recognition using deep learning method. CNNs have been the most used deep learning method for character recognition and have proved best results for many scripting languages such as Chinese, Arabic, Japanese, and many more. Various character datasets and their details were also presented. The implementation of character recognition in Keras and TensorFlow platform was explained with Tamil offline handwritten character recognition as example.

References

1. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE. **86**(11), 2278–2324 (1998)
2. LeCun, Y., Boser, B.E., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W.E., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems (pp. 396–404) (1990).
3. Ciregan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In 2012 IEEE conference on computer vision and pattern recognition (pp. 3642–3649). IEEE (2012, June)
4. 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)
5. LeCun, Y., Cortes, C., Burges, C. J: The MNIST database of handwritten digits, 1998. URL http://yann.lecun.com/exdb/mnist, 10, 34 (1998)
6. Grother, P. J.: NIST special database 19 handprinted forms and characters database. National Institute of Standards and Technology (1995)
7. Cohen, G., Afshar, S., Tapson, J., Van Schaik, A.: EMNIST: extending MNIST to handwritten letters. In 2017 international joint conference on neural networks (IJCNN) (pp. 2921–2926). IEEE (2017, May)
8. Liu, C. L., Yin, F., Wang, D. H., Wang, Q. F.: CASIA online and offline Chinese handwriting databases. In 2011 international conference on document analysis and recognition (pp. 37–41). IEEE (2011, September)
9. Liu, C.L., Yin, F., Wang, D.H., Wang, Q.F.: Online and offline handwritten Chinese character recognition: benchmarking on new databases. Pattern Recogn. **46**(1), 155–162 (2013)
10. Zhang, X.Y., Bengio, Y., Liu, C.L.: Online and offline handwritten Chinese character recognition: a comprehensive study and new benchmark. Pattern Recogn. **61**, 348–360 (2017)
11. Yin, F., Wang, Q. F., Zhang, X. Y., Liu, C. L.: ICDAR 2013 Chinese handwriting recognition competition. In 2013 12th international conference on document analysis and recognition (pp. 1464–1470). IEEE (2013, August)
12. Lawgali, A., Angelova, M., Bouridane, A.: HACDB: handwritten Arabic characters database for automatic character recognition. In European workshop on visual information processing (EUVIP) (pp. 255–259). IEEE (2013, June)
13. Boufenar, C., Kerboua, A., Batouche, M.: Investigation on deep learning for off-line handwritten Arabic character recognition. Cogn. Syst. Res. Cognitiv. Syst. Res. **50**, 180–195 (2018)
14. El-Sawy, A., Loey, M., Hazem, E.B.: Arabic handwritten characters recognition using convolutional neural network. WSEAS Trans Computer Res. **5**, 11–19 (2017)
15. Rabby, A.S.A., Haque, S., Islam, M.S., Abujar, S., Hossain, S.A.: Ekush: a multipurpose and multitype comprehensive database for online off-line Bangla handwritten characters. In: International conference on recent trends in image processing and pattern recognition, pp. 149–158. Springer, Singapore (2018, December)
16. Biswas, M., Islam, R., Shom, G.K., Shopon, M., Mohammed, N., Momen, S., Abedin, A.: Banglalekha-isolated: a multi-purpose comprehensive dataset of handwritten Bangla isolated characters. Data Brief. **12**, 103–107 (2017)
17. Sarkar, R., Das, N., Basu, S., Kundu, M., Nasipuri, M., Basu, D.K.: CMATERdb1: a database of unconstrained handwritten Bangla and Bangla-English mixed script document image. Int. J. Doc. Analys. & Recog. (IJDAR). **15**(1), 71–83 (2012)
18. Indian Statistical Institute ISI Handwritten Character Database. https://www.isical.ac.in/~ujjwal/download/database.html. Cited 1 May 2020
19. Bhattacharya, U., Shridhar, M., Parui, S.K., Sen, P.K., Chaudhuri, B.B.: Offline recognition of handwritten Bangla characters: an efficient two-stage approach. Pattern. Anal. Applic. **15**(4), 445–458 (2012)

20. Rahman, M.M., Akhand, M.A.H., Islam, S., Shill, P.C., Rahman, M.H.: Bangla handwritten character recognition using convolutional neural network. International Journal of Image, Graphics and Signal Processing (IJIGSP). **7**(8), 42–49 (2015)
21. Karatzas, D., Gomez-Bigorda, L., Nicolaou, A., Ghosh, S., Bagdanov, A., Iwamura, M., … Shafait, F.: ICDAR 2015 competition on robust reading. In 2015 13th international conference on document analysis and recognition (ICDAR) (pp. 1156–1160). IEEE (2015, August)
22. De Campos, T., Bodla, R. B., Varma, M.: The chars74k dataset (2009)
23. Veit, A., Matera, T., Neumann, L., Matas, J., Belongie, S.: Coco-text: dataset and benchmark for text detection and recognition in natural images. arXiv preprint arXiv:1601.07140 (2016)
24. Yuan, T. L., Zhu, Z., Xu, K., Li, C. J., & Hu, S. M: Chinese text in the wild. arXiv preprint arXiv:1803.00085 (2018)
25. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Ghemawat, S.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
26. HpLabs Isolated Handwritten Tamil Character dataset, http://lipitk.sourceforge.net/datasets/tamilchardata.htm
27. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J Machine Learn Res. **15**(1), 1929–1958 (2014)
28. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
29. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
30. Vamsidhar, E., Kanagachidambaresan, G.R., Prakash, K.B.: Application of machine learning and deep learning, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_8

Keras and TensorFlow: A Hands-On Experience

Ferdin Joe John Joseph, Sarayut Nonsiri, and Annop Monsakul

Introduction

TensorFlow architecture is explained in such a way to make the reader understand why it is needed to learn as a prerequisite for Keras [1]. TensorFlow and its supporting libraries in Python are explained for this purpose. The understanding of interoperability of these libraries is required to understand TensorFlow. This chapter is organized as below:

1. TensorFlow architecture
2. Introduction to Keras
3. Installation of TensorFlow and Keras in Jupyter Notebooks – hardware aspects
4. Installation of TensorFlow and Keras in Jupyter Notebooks – software aspects
5. Linear regression using Keras – case study
6. Binary classification using Keras – case study
7. Multiclass classification – case study

After obtaining prerequisite, Keras is presented. With TensorFlow as backend, Keras is explained. The theoretical aspects of Keras are explained. This is followed by the operational functionalities of the Keras library for deep learning applications. The operational functionalities of Keras are explained with the help of Python code snippets. These are the theoretical aspects of Keras.

To work with Keras in deep learning, a section on how to install and configure the library is needed. This includes both hardware and software aspects. Google

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-66519-7_12

F. J. John Joseph (✉) · S. Nonsiri · A. Monsakul
Faculty of Information Technology, Thai – Nichi Institute of Technology, Bangkok, Thailand
e-mail: ferdin@tni.ac.th

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
Corrected Publication 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_4

Colab uses TPUs in the cloud, but for standalone versions, GPUs are needed. The list of GPUs supported by TensorFlow and Keras are listed. This is added with other hardware requirements and drivers needed to be installed including NVIDIA software. The installation of TensorFlow and Keras in Jupyter Notebooks' hardware aspects provides enough information on how to prepare a PC or laptop in order to make it compatible enough to work with TensorFlow and Keras. This part of the chapter will give a step-by-step procedure on how to make the GPUs active in a PC or laptop and enable CUDA processing units.

Once the hardware of the system is made compatible for TensorFlow and Keras, the software has to be installed. The interoperability between GPU and Keras with TensorFlow is explained in this section. This includes a step-by-step process on how to install Keras in TensorFlow. The steps include the command prompt statements needed to create an environment in Anaconda Navigator and Jupyter Notebooks. After the environment is created, a step-by-step process on how to make the GPUs sync with the environment set with Anaconda Navigator is illustrated. After making all the synchronizations positive, the libraries of Keras in TensorFlow are explained in a step-by-step process and screenshots. After installing Keras, the validity of the environment and libraries is checked using some TensorFlow library functions.

After the installation of Keras is made successful, case studies are presented for explaining how to code using TensorFlow and Keras. Linear regression and binary and multiclass classification are explained with examples as case study. Jupyter Notebooks are shared, and the code snippets are explained. This includes importing data from external sources, including library data like MNIST dataset, etc. The data imported are subjected to preprocessing and setting up of CNN parameters like hidden, convolution, and max pooling layers. The output of these case studies are explained with the code and the epochs done in each training execution.

Linear regression is presented with diamond price prediction. This diamond price dataset is in CSV format. This dataset contains 54,000 diamond records. Pandas library is used to explain the usage of the dataset in CSV format. Artificial neural network (ANN) algorithm is used for creating a prediction model.

Binary classification is presented with cat or dog classification that involves predicting the images as containing either a dog or a cat. The experiment section is to demonstrate using deep learning convolutional neural networks with free Kaggle dataset. The training data contains 25,000 images, and the testing data contains 12,500 images of dogs and cats. The accuracy is measured, and the confusion matrix is created and visualized using Seaborn library.

Multiclass classification is presented with the MNIST database of handwritten digits available from inbuilt Keras dataset, which is a standard dataset used in computer vision. This database contains 28 × 28 gray scale images of the 10 digits. There are 70,000 images for training, whereas 10,000 images excluding the trained images are taken for testing. For the experiment section, multilayer perceptron (MLP) with convolutional neural network (CNN) algorithms are compared in terms of accuracy. All these applications are explained with domain knowledge and source code. The GitHub link for the source code files is shared in the link [2] below.

https://github.com/ferdinjoe/Chapter-4-Working-with-Keras

1 TensorFlow Architecture

Prior to exploring Keras, TensorFlow needs to be discussed in detail. TensorFlow is the most eminent libraries developed by Google for machine learning and deep learning applications. This library is built to run on CPUs, GPUs, IoT processors like Jetson Nano, neural network stick, etc. This library was first made public in 2015. However, a stable version is made available since 2017 under the Apache Open Source License. Libraries in Apache Open Source License are allowed to be used, modified, and redistributed without any royalty to Google.

The architecture of TensorFlow consists of three parts as given in the diagram below (Fig. 1).

The input fed into TensorFlow is a multidimensional array, and these arrays are called tensors. Flow of a multidimensional array gave the name TensorFlow. These tensors go into the system as input in one end and go through various operations and finally produce output.

In Python TensorFlow is activated using the following import statements more prominently.

```
import tensorflow as tf
```

The notation "*tf*" is widely used by developers to mention the usage of TensorFlow in the code.

This has two phases:

1. Development phase
2. Run phase

Development phase is for training data and is performed in high-performance workstations, PCs, or laptops with the help of GPUs. Run phase is done on any kind of machine including desktop, cloud, or mobile devices.

The list of algorithms in TensorFlow 2.0 is given below.

```
from keras.models import Sequential
model = Sequential()
```

After extensive usage of TensorFlow by researchers and data scientists, Keras was developed on top of TensorFlow.

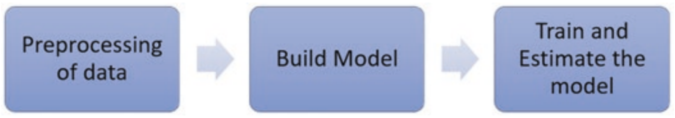


Fig. 1 TensorFlow architecture

2 Introduction to Keras

Keras is a Python-based neural network API which is used to run TensorFlow, CNTK, and Theano. It is used with TensorFlow 2.0 on top of which the latest version of Keras and version – Keras 2.3.0 – is embedded for the source code listed in this chapter. In Python, Keras is used with the notation tf.keras. It has four working principles as follows:

1. User friendliness
2. Modularity
3. Easy extensibility
4. Work with Python

Models are the most important data structures in Keras. Sequential is a model which is widely used. The procedure to use the sequential model is given below:

```
tf.keras.layers.Dense(units,activation=None,use_bias=True,
                      kernel_initializer="glorot_uniform",
                      bias_initializer="zeros",
                      kernel_regularizer=None,
                      bias_regularizer=None,
                      activity_regularizer=None,
                      kernel_constraint=None,
                      bias_constraint=None, **kwargs)
```

Types of Keras Layers
There are 11 different types of Keras layers available to configure the deep learning neural network and are listed as below:

1. **Core Layers**

   Dense
   It is used as a core layer in the densely connected NN layer.

```
      keras.layers.Activation(activation)
```

The above code snippet gets *n*D tensor and gives *n*D tensor as output. The input takes *n*D batch size with its dimensions and batch size. This input after passing through this core layer produces an output of the same size as input.

Activation
This applies an activation function to the given input. The input and output are of the same size. The code snippet below shows the usage of this layer function.

```
keras.layers.Dropout (rate, noise_shape=None, seed=None)
```

Dropout

The input layer is given with the transformation of dropout.

```
keras.layers.Flatten(data_format=None)
```

Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

Flatten

```
keras.layers.Reshape(target_shape)
```

It is used to flatten the input, and it does not affect the size of input and output. This layer is normally used in the layer corresponding to the model of the data trained and tested.

Reshape

Reshape is used as a layer to convert the shape of input tensor to the size of output tensor.

```
keras.layers.Permute(dims)
```

Permute

This layer uses a permutation to change the dimension of the input. It is used in scenarios like connecting recurrent neural networks (RNN) and convolution networks (ConvNets).

```
keras.layers.RepeatVector(n)
```

RepeatVector

Repeats the input vector n number of times.

```
keras.layers.Lambda(function, output_shape=None, mask=None, arguments=None)
```

Lambda

Lambda is used to wrap arbitrary expression as layer object.

```
keras.layers.ActivityRegularization(l1=0.0, l2=0.0)
```

Activity Regularization

Activity regularization is used to apply an update to the cost function based on input activity.

```
conda create --name tensorflow --clone root
```

Similarly, there are some other core layers like masking and spatial dropout 1D, 2D, and 3D.

2. **Convolutional Layers**

As same as TensorFlow, the convolutional layers in Keras consist of Conv 1D, Conv 2D, Separable Conv 1D, Separable Conv 2D, Depthwise Conv 1D, Depthwise Conv 2D, Conv 2D Transpose, and 3D of all the above.

3. **Pooling Layers**

The pooling layers of Keras are as follows: max pooling, average pooling, average max pooling, global max pooling, etc.

4. **Locally Connected Layers**

Locally connected 1D and locally connected 2D are layers present in locally connected layers.

5. **Recurrent Layers**

Recurrent layers consist of those functional exclusively with recurrent neural networks (RNN), gated recurrent unit (GRU), and long short-term memory (LSTM) layers.

6. **Embedding Layers**
7. **Merge Layers**
8. **Advanced Activation Layers**

These are usual activation layers but developed for special cases. LeakyReLU is a rectified version of the rectified linear unit, whereas PReLU is a parametric variation of ReLU. In addition to ReLU and softmax, exponential linear unit (ELU) and threshold ReLU are also available with Keras.

9. **Normalization Layers**

BatchNormalization is a normalization layer provided in Keras.

10. **Noise Layers**

Gaussian noise, Gaussian dropout, and alpha dropout are the noise layers which are used to reduce the noise in data provided.

11. **Layer Wrappers**

Time distributed wrapper is used to apply to the time space of the input, whereas bidirectional wrapper is used with RNNs.

3 Installation of TensorFlow and Keras in Jupyter Notebooks: Hardware Aspects

The ideal configuration of Keras normally requires the following in a Windows 10 OS.

```
NVIDIA GeForce GTX 1060 6GB GDDR5 and above
Core i7-7700 HQ and above or i5 10th generation
16GB DDR4 RAM and above
```

Keras runs on top of TensorFlow. So the hardware dependencies are the same as TensorFlow. NVIDIA GPUs are normally recommended for Keras installation. Brands other than NVIDIA are yet to support Keras and TensorFlow. This is due to the interdependency of CUDA middleware between TensorFlow and NVIDIA GPU.

Install the required drivers for NVIDIA GeForce GTX or RTX to enable the GPU to sync with the operating system. A GPU version of TensorFlow has to be installed. Install the CUDA toolkit downloaded from https://developer.nvidia.com/cuda-toolkit. Follow readme section in the GitHub link.

Once the environment variables are set, restart the machine. After restarting the machine, follow the steps in Sect. 4.4.

Fig. 2 Software needed to install Keras

4 Installation of TensorFlow and Keras in Jupyter Notebooks: Software Aspects

The installation of Keras involves the following software (Fig. 2).

Anaconda Navigator has to be installed by downloading it from https://www.anaconda.com/download. Jupyter Notebook with Python distribution is installed by default. There are two ways of installing TensorFlow and Keras:

1. Pip from the source.
2. Install in a virtual environment with Jupyter Notebook in Anaconda Navigator.

The tutorial in this chapter involves Jupyter Notebook in Anaconda Navigator.

On top of all the above installations, install Visual Studio 2015. Now restart the machine again. The installation of hardware and software support is now done.

After restarting the machine, follow the steps below to synchronize GPU with Keras and TensorFlow:

1. Installation of environment for TensorFlow

   Open Anaconda Prompt, enter the following command:

```
activate tensorflow
pip install --ignore-installed --upgrade tensorflow-gpu
```

– clone root will inherit the libraries from default Python to TensorFlow environment.

2. Installation of TensorFlow with GPU version

   Enter the following commands:

```
call Activate Tensorflow
```

3. TensorFlow validation

   Open Anaconda Prompt and type

```
import tensorflow as tf
print(tf.__version__)
```

Then open Anaconda Navigator and type

```
activate tensorflow environment
```

The response for the above two statements must show the version of TensorFlow installed.

4. Installation and validation of Keras
   In Anaconda Prompt, open TensorFlow environment and type

```
conda install keras
```

Now enter

```
import keras
Using TensorFlow backend.
print(keras.__version__)
```

After installation of Keras is complete, the library functionality could be checked using the statements below.

```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

2.1.6

5. Check whether Keras and TensorFlow are running on GPU

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 14869837826445014983
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 7103961498
locality {
  bus_id: 1
  links {
  }
}
incarnation: 9950417734109268282
physical_device_desc: "device: 0, name: GeForce GTX 1070 Ti, pci bus id: 0000:01:00.0,
compute capability: 6.1"
]
```

The output for this statement will be something like the one below.

```
from keras.datasets import boston_housing
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

In the absence of GPU or improper GPU synchronization, Keras and TensorFlow will activate in CPU mode, and it will run slower than GPU.

5 Linear Regression Using Keras: Case Study

In this section, we will explore how to create a linear regression model using Keras to determine the relationship between the independent input variables and the dependent output variable. The Boston housing dataset is used to demonstrate the training of a model in order to make predictions of a house price value in Boston, Massachusetts, area. This dataset [3] was collected by the US Census Service. It can be accessed from the StatLib archive (http://lib.stat.cmu.edu/datasets/boston), which is maintained by Carnegie Mellon University (CMU). There are 506 records, and each contains independently 13 features of houses and 1 target variable (MEDV).

1. Load the Boston housing dataset using Keras

For loading the Boston housing dataset, Keras provides a *load_data()* method to download the dataset directly from the keras.dataset module. The method returns two tuples that contain training (*x_train, y_train*) and testing data (*x_test, y_test*). This method is beneficial for the users because they do not need to use a CSV file downloaded from other sources.

```
import pandas as pd
attributes_name = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age',
                   'dis', 'rad', 'tax', 'ptratio', 'black', 'lstat']
df = pd.DataFrame(x_train)
df.columns = attributes_name
df['medv'] = y_train
df.head(10)
```

Pandas is an open-source Python library for data manipulation and analysis. In this example, it is used to explore the training samples in a tabular format. The first line of code is importing a pandas library called as pd. in order to refer to any pandas functions within a program. *x_train variable* is loaded into the pandas DataFrame object. Since there is no available column name from a built-in Keras dataset, this variable is loaded. The attributes_name variable contains a list of attribute names and assigned to DataFrame column names (*df.columns*). The y_train, as the target variable, is added into a new DataFrame column. The *df.head(10)* method returns the first ten rows of the training data.

```
print('Training data:', x_train.shape)
print('Testing data:', x_test.shape)
```

For the Boston housing dataset, there are 506 records. This information is splitting into two parts for training and testing purposes, as shown above. The training data (*x_train*) contains 404 records, and the testing data (*x_test*) contains 102 records. The *x_train.shape* and *x_test.shape* commands return the number of training and testing data dimensions in (rows, columns) format. The result shows that the training data shape is (*404, 13*), and the testing data shape is (*102, 13*). It means that there are 404 rows and 13 columns for training data, and there are 102 rows and 13 columns for testing data.

```
from keras.models import Sequential
from keras.layers import Dense
```

```
Training data: (404, 13)
Testing data: (102, 13)
```

The necessary modules are imported into the Python environment that contains the relevant functions, for example, *Sequential* and *Dense* modules.

```
model = Sequential()
model.add(Dense(14, input_dim=13, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

From the following code block, it is a step of defining and compiling a model. A Sequential class allows us to build a neural network model by stacking the different layers using the *add()* method. A *Sequential()* constructor is instantiated to a model variable. A Dense class is a fully connected neural network layer in which every input is connected to every output.

For the first dense layer in a model, the input dimension needs to be specified by setting the input_dim parameter the same size as the number of columns of *x_train* and *x_test* variables [4]. In this example, there are 13 features. 16 is the number of neurons in this layer. ReLU ("*relu*") is selected as an activation function. Hence, there are three parameters passed into this layer. The second layer is defined by passing two parameters: the six neurons and ReLU ("*relu*") activation function. The last layer is an output layer in which there is only one neuron, and the activation function is linear.

A *compile()* method is specifying the learning process. The mean squared error (“*mean_squared_error*”) is selected as a loss function, and the optimizer parameter is the “adam” algorithm. As a result, this simple architecture consists of three layers. A *summary()* method is to display the neural network architecture, such as the number of layers, the output shape, and the number of parameters.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 14)                196
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 90
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 7
=================================================================
Total params: 293
Trainable params: 293
Non-trainable params: 0
_________________________________________________________________
```

The following results are returned:

```
history = model.fit(x_train, y_train, epochs=1000)
```

A fit() method is used to train the regression model using the training data (*x_train*) and target data (*y_train*). Additionally, an epochs parameter is the number of iterations that passed all training data through the neuron network model forward and backward in order to update the weights. In this example, the epochs parameter is set to 1000.

```
Epoch 1000/1000
404/404 [==============================] - 0s 59us/step - loss: 20.2829
```

During the training process, the training statistics is displayed, as shown below:

```
performance = model.evaluate(x_test, y_test)
print(performance)
```

Evaluate() method is to access the trained model performance using testing data (*x_test* and *y_test*). The lower loss value is a better performance. As a result, the performance is approximately 29.27.

```
102/102 [==============================] - 0s 205us/step
29.270214753992416
```

```
102/102 [==============================] - 0s 205us/step
29.270214753992416
```

From all preceding the code blocks, it is implementing a simple neuron network architecture. However, this trained model performance can be improved. In the following sections, there are several techniques introduced that minimize the loss function value, such as rescaling data, regularization, batch normalization, and dropout. These techniques can provide a better result.

Rescaling data is one of the techniques used to overcome that the dataset value has different ranges. This issue might cause some more significant range features to have a greater impact on more than a lesser range of features. Therefore, we need to rescale both training and testing data before training a model.

Scikit-learn is an open-source machine learning library in Python, but we will use it for preprocessing data purposes. A *fit_transform()* method from *MinMaxScaler()* class is used to transform each feature between 0 and 1 for both training (*x_train*) and testing (*x_test*) data.

```
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)
```

From the code below, using the pandas library to explore the training samples after rescaling data in a tabular format, the following results are returned with all features rescaled between 0 and 1.

```
attributes_name = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age',
                   'dis', 'rad', 'tax', 'ptratio', 'black', 'lstat']
df = pd.DataFrame(x_train)
df.columns = attributes_name
df['medv'] = y_train
df.head(10)
```

This neural network is designed to perform a regression analysis. The additional Keras modules are imported to prevent overfitting issues, such as *regularizers*, *Dropout*, and *BatchNormalization*. Therefore, the loss value can significantly lower than the previous regression model.

kernel_regularizer parameter can be used to apply a penalty on the layer's kernel. We can add regularization to the Keras layers. The default is None. In this example, *keras.regularizers.l2()* method is added to the Dense layer. This method is a l2 regularization type with regularization strength between 0 and 1. In this example, l is 0.1. A too high regularization strength value may cause the model to underfit. If l1 regularization is needed, just only replace keras.regularizers.l2() with keras.regularizers.l1() method with desired strength value.

```
model.add(Dense(14, input_dim=13,
                    activation='relu',
                    kernel_regularizer=regularizers.l2(0.1)))
```

The dropout technique is randomly dropping out the neurons on input and hidden layers during training based on a fraction rate. This technique helps the model generalize enough to fit on unseen data. In this example, the dropout layer is added to the model, and the fraction rate is 0.01, as shown below:

```
model.add(Dropout(.01))
```

BatchNormalization() method is a technique used to reduce a large output on the hidden layer and help speed up the training process. For adding batch normalization layer into the model, model.add() method is simply used as follows:

```
model.add(BatchNormalization())
```

The new model is defined and compiled, and a model summary is as follows:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras import regularizers
from keras.layers.normalization import BatchNormalization

model = Sequential()
model.add(Dense(14, input_dim=13, activation='relu',
                kernel_regularizer=regularizers.l2(0.1)))
model.add(BatchNormalization())
model.add(Dropout(.01))
model.add(Dense(6, activation='relu', kernel_regularizer=regularizers.l2(0.1)))
model.add(BatchNormalization())
model.add(Dropout(.01))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 14)                196
_________________________________________________________________
batch_normalization_1 (Batch (None, 14)                56
_________________________________________________________________
dropout_1 (Dropout)          (None, 14)                0
_________________________________________________________________


dense_5 (Dense)              (None, 6)                 90
_________________________________________________________________
batch_normalization_2 (Batch (None, 6)                 24
_________________________________________________________________
dropout_2 (Dropout)          (None, 6)                 0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 7
=================================================================
Total params: 373
Trainable params: 333
Non-trainable params: 40
```

The following code is to fit a model by splitting training data into two parts: training and validation data. The validation_split parameter is set to 0.1 meaning the number of records in training data is 90%, and validation data is 10%.

```
history = model.fit(x_train, y_train, epochs=1000, validation_split=0.1, verbose=2)
```

The following results are returned:

```
Epoch 1000/1000 - 0s - loss: 10.8459 - val_loss: 4.9475
```

Matplotlib library is an open-source visualization tool in Python. To display the graph in Jupyter Notebook, a *%matplotlib inline* command allows us to embed an output graph in the notebook. The *pyplot* module is used for plotting the line graph by the given x and y variables. "*import matplotlib.pyplot as plt*" is importing a *pyplot* module from the Matplotlib library and naming it as plt. In this example, it is used to display a graph to compare between loss and validation loss values as follows:

```
%matplotlib inline
import matplotlib.pyplot as plt
loss=history.history['loss']
val_loss=history.history['val_loss']
epochs=range(len(loss))
plt.plot(epochs, loss, label='Training Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.title('training loss and validation loss')
plt.legend(loc=0)
plt.savefig('Training loss vs. Validation loss.png', dpi=150)
plt.figure()
```

The preceding code returns the result, as shown in Fig. 3.

model.evaluate() method is used to evaluate the performance of the model using testing data.

```
performance = model.evaluate(x_test, y_test)
print(performance)
```

model.predict() method is used to predict the testing dataset using the trained model.

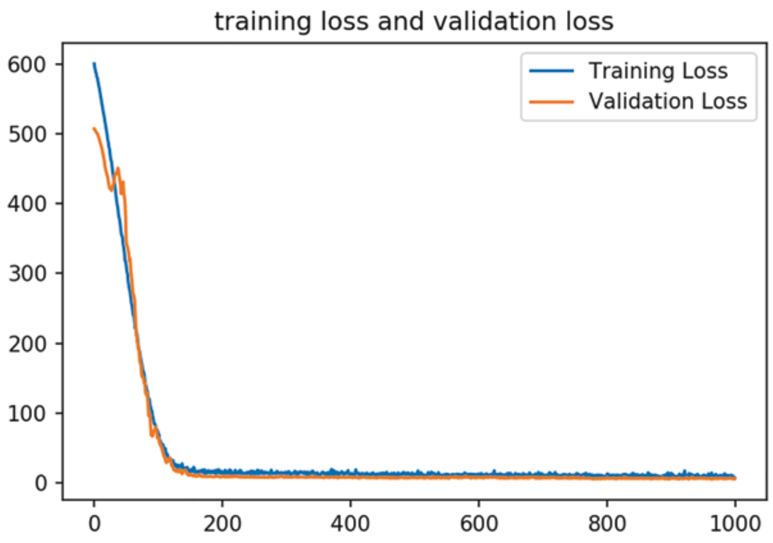


Fig. 3 Training loss vs. validation loss

```
y_pred = model.predict(x_test)
print(y_pred)
```

6 Binary Classification Using Keras: Case Study

This section will explore how to build a deep neural network model to classify cat and dog images using Keras [5]. The dataset used in this section can be downloaded from the link: https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip. The file size is about 66 MB. There are 2000 training images and 1000 validation images.

In this first step, the libraries and modules are imported from Keras into the Python environment.

```
import keras
from keras import Model
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.layers.convolutional import Conv2D, MaxPooling2D
import os
```

This step, a *cats_and_dogs_filtered* folder is set as a root path and contains the folders for training and validation image directories. Each folder contains cat and dog images subfolders for creating a model.

```
root_path = 'cats_and_dogs_filtered'

train_folder = os.path.join(root_path, 'train')
validation_folder = os.path.join(root_path, 'validation')

train_dogs_folder = os.path.join(train_folder, 'dogs')
train_cats_folder = os.path.join(train_folder, 'cats')

validation_dogs_folder = os.path.join(validation_folder, 'dogs')
validation_cats_folder = os.path.join(validation_folder, 'cats')
```

The below code is to define the cat and dog filenames in the training and validation directories. The result shows the first five filenames of each folder.

```
train_dog_filenames = os.listdir(train_dogs_folder)
train_cat_filenames = os.listdir(train_cats_folder)
validation_dog_filenames = os.listdir(validation_dogs_folder)
validation_cat_filenames = os.listdir(validation_cats_folder)

print('Train filenames:')
print(train_dog_filenames[:5])
print(train_cat_filenames[:5])

print('Validation filenames:')
print(validation_dog_filenames[:5])
print(validation_cat_filenames[:5])
```

```
Train filenames:
['dog.0.jpg', 'dog.1.jpg', 'dog.10.jpg', 'dog.100.jpg',
  'dog.101.jpg']
['cat.0.jpg', 'cat.1.jpg', 'cat.10.jpg', 'cat.100.jpg',
  'cat.101.jpg']

Validation filenames:
['dog.2000.jpg',     'dog.2001.jpg',     'dog.2002.jpg',
  'dog.2003.jpg', 'dog.2004.jpg']
['cat.2000.jpg',     'cat.2001.jpg',     'cat.2002.jpg',
  'cat.2003.jpg', 'cat.2004.jpg']
```

The preceding code is to check the number of cat and dog images.

```
dogs_num = len(train_dog_filenames)
cats_num = len(train_cat_filenames)

print('training dog:', dogs_num)
print('training cat:', cats_num)

val_dogs_num = len(validation_dog_filenames)
val_cats_num = len(validation_cat_filenames)

print('validation dog:', val_dogs_num)
print('validation cat:', val_cats_num)
```

```
training dog: 1000
training cat: 1000
validation dog: 500
```

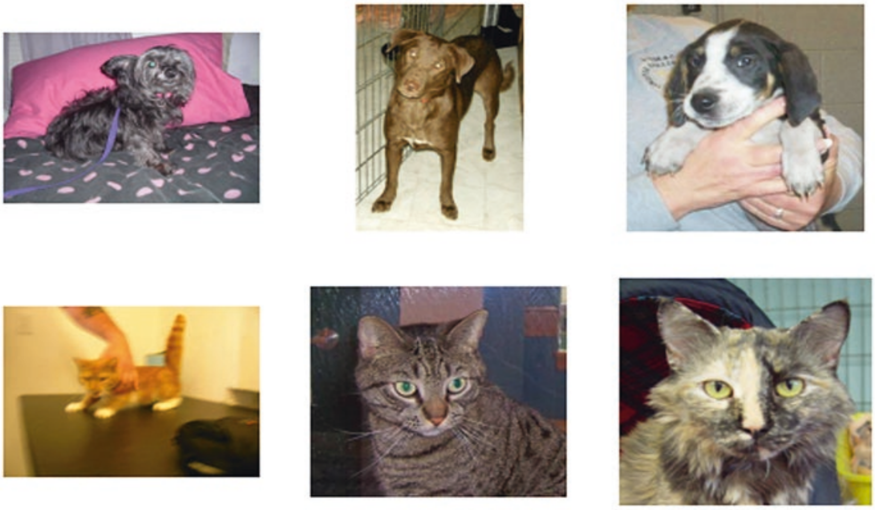


Fig. 4 Cat and dog samples

```
validation cat: 500
```

This is plotting the first three of cat and dog images using the Matplotlib module (Fig. 4).

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

dog_path = train_dogs_folder+'\\'
cat_path = train_cats_folder+'\\'

plt.subplot(231)
plt.imshow(mpimg.imread(dog_path+train_dog_filenames[0]))
plt.axis('Off')
plt.subplot(232)
plt.imshow(mpimg.imread(dog_path+train_dog_filenames[1]))
plt.axis('Off')
plt.subplot(233)
plt.imshow(mpimg.imread(dog_path+train_dog_filenames[2]))
plt.axis('Off')
plt.subplot(234)
plt.imshow(mpimg.imread(cat_path+train_cat_filenames[0]))
plt.axis('Off')
plt.subplot(235)
plt.imshow(mpimg.imread(cat_path+train_cat_filenames[1]))
plt.axis('Off')
plt.subplot(236)
plt.imshow(mpimg.imread(cat_path+train_cat_filenames[2]))
plt.axis('Off')
plt.savefig('Dog Cat.png', dpi=100)
plt.show()
```

This preceding code is to perform the data augmentation and rescale for the training and validation images using *ImageDataGenerator* class.

```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1.0/255.,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True)
validation_datagen  = ImageDataGenerator(rescale = 1.0/255.)
```

This step is to define the training and validation folders using the *flow_from_directory()* method. The batch size is set to 100, as it is the number of training images in a single batch. "*binary*" is assigned to *class_mode* parameter due to this being a binary classification problem. *target_size* parameter is to set the image height and width. In this setting, each image is 150 × 150 pixels.

```
train_generator = train_datagen.flow_from_directory(train_folder,
                                                    batch_size=100,
                                                    class_mode='binary',
                                                    target_size=(150, 150))

validation_generator = train_datagen.flow_from_directory(validation_folder,
                                                    batch_size=100,
                                                    class_mode='binary',
                                                    target_size=(150, 150))
```

Building a binary classifier by defining, compiling, and training the model.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
history = model.fit(train_generator, validation_data=validation_generator, epochs=1000,
verbose=2)
```

The validation accuracy is returned (82%) as shown in the following results:

```
Epoch 999/1000 - 9s - loss: 0.0104 - accuracy: 0.9970 - val_loss: 1.2515 - val_accuracy: 0.7960
Epoch 1000/1000 - 9s - loss: 0.0094 - accuracy: 0.9960 - val_loss: 1.1365 - val_accuracy: 0.8200
```

After training the binary classifier model, we will test a new image unseen before [6] and then load and preprocess a new image (Fig. 5) for predicting using *predict()* function.

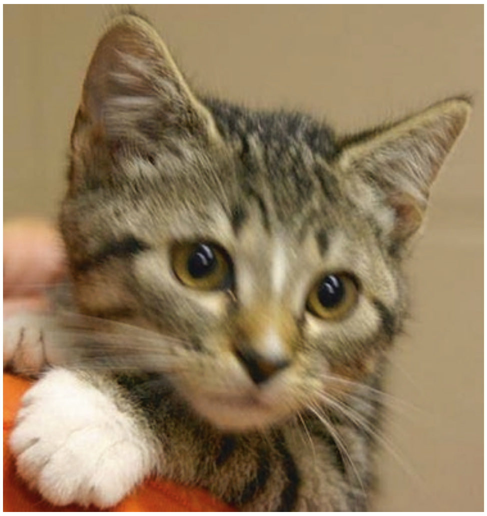


Fig. 5 Cat.jpg

```
from keras.preprocessing import image
import numpy as np

# load a new image
filename = 'cat.jpg'
img = image.load_img(filename, target_size=(150, 150))

# preprocess a new image
img = image.img_to_array(img)
img = np.expand_dims(img, axis=0)

# predict a new image
predicted_result = model.predict(img)
print(predicted_result[0][0])
if predicted_result[0][0] == 0:
    result = 'This image is a Cat.'
else:
    result = 'This image is a Dog.'
print(result)
```

The preceding code returns the result

```
[0.]
This image is a Cat.
```

The train_generator.class_indices attribute returns a dictionary that contains class names and indices. For example, {“cats”: 0, “dogs”: 1}. It means that if predicted_result[0][0] is 0, then the predicted image is a cat. Otherwise, the image is a dog. As a result, its predicted_result[0][0] is *[0.]*, and it prints “*This image is a Cat*.” Finally, the model can predict the image correctly.

7 Multiclass Classification: Case Study

In this section, we will introduce a multiclass classification on the MNIST (Modified National Institute of Standards and Technology) dataset using convolutional neural network (CNN) in Keras. The MNIST is a handwritten digits dataset [7]. There are 60,000 training images and 10,000 testing images. Each image is gray scale 28 x 28 pixels.

Importing the libraries, modules, and MNIST dataset.

```
import keras
from keras.datasets import mnist
from keras import Model
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.layers.convolutional import Conv2D, MaxPooling2D
```

mnist.load_data() method is used to load the MNIST dataset from the Keras module. The result returns the training (*x_train, y_train*) and testing (*x_test, y_test*) data.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The *x_train.shape* and *x_test.shape* commands return the number of training and testing data dimensions in the number of images and the number of pixels format. The result shows that the training data shape is (*60,000, 28, 28*), and the testing data shape is (*10,000, 28, 28*). It means that the training data (*x_train*) contains 60,000 images, and the testing data (*x_test*) contains 10,000 records, and each image dimension is 28 x 28 pixels.

```
print('Train image:', x_train.shape)
print('Test image:', x_test.shape)
```

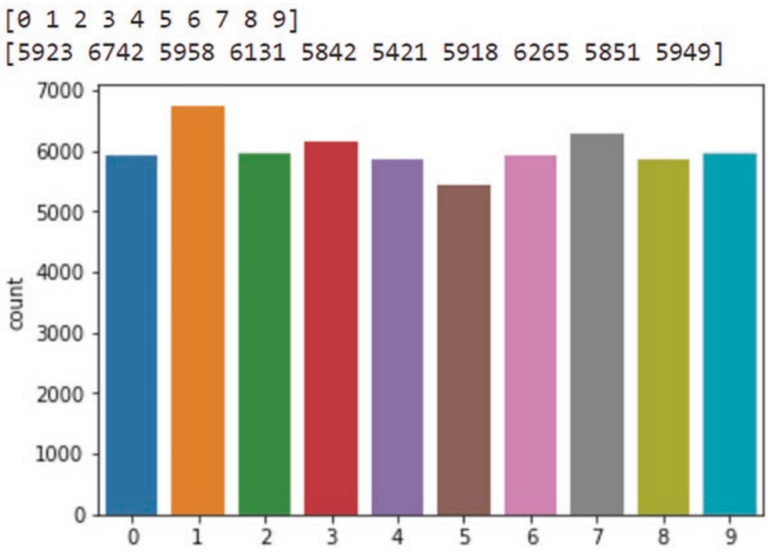
```
Train image: (60000, 28, 28)
Test image: (10000, 28, 28)
```

Seaborn and NumPy libraries are imported to explore the distribution of the handwritten digit number samples.

```
import seaborn as sns
import numpy as np
(number, counts) = np.unique(y_train, return_counts=True)
g = sns.countplot(y_train)
print(number)
print(counts)
```

The preceding code returns the result, as shown in Fig. 6.

This code block is to plot an image using the Matplotlib library, as shown in Fig. 7.



```
[0 1 2 3 4 5 6 7 8 9]
[5923 6742 5958 6131 5842 5421 5918 6265 5851 5949]
```

Fig. 6 Training data plot

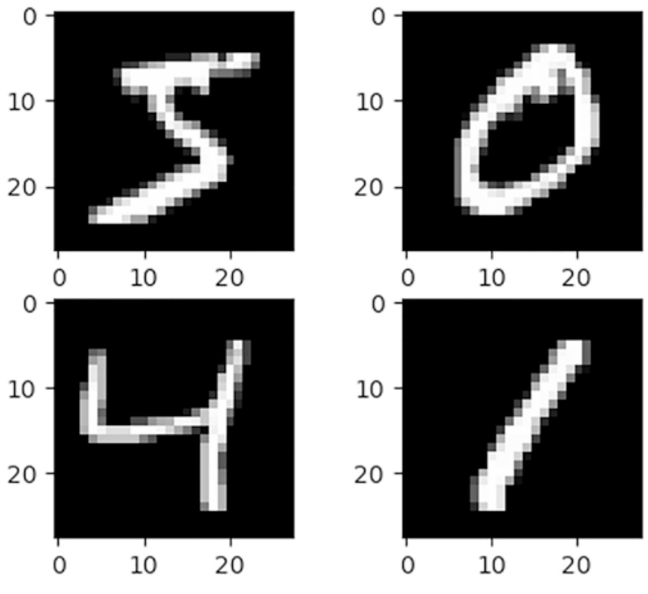


Fig. 7 Handwritten digits samples

```
import matplotlib.pyplot as plt
plt.subplot(221)
plt.imshow(x_train[0], cmap='gray')
plt.subplot(222)
plt.imshow(x_train[1], cmap='gray')
plt.subplot(223)
plt.imshow(x_train[2], cmap='gray')
plt.subplot(224)
plt.imshow(x_train[3], cmap='gray')
plt.savefig('Number.png', dpi=100)
plt.show()
```

The image result is returned as follows:
Building a simple neural network model by defining and compiling.

```
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

The training data is used to train the CNN model.

```
history = model.fit(x_train, y_train, epochs=1000)
```

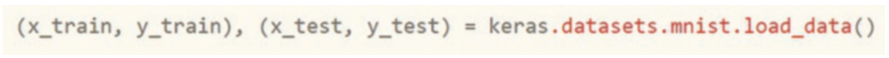
Evaluate the model performance using model.evaluate() method.

```
v_loss, v_acc = model.evaluate(x_test, y_test)
print(v_loss, v_acc)
```

The following section discusses on how to build a convolutional neural network model. The relevant modules are imported.

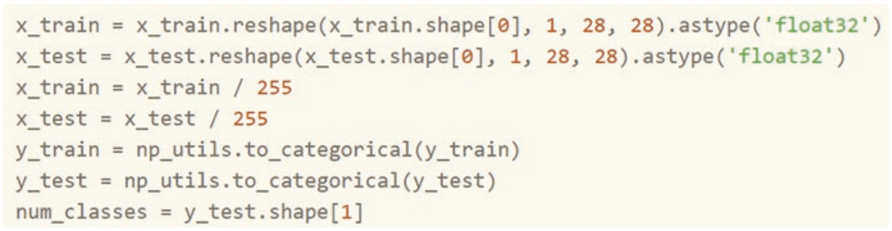
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.utils import np_utils
```

mnist.load_data() method is used to load the MNIST dataset from the Keras module. The result returns the training (*x_train, y_train*) and testing (*x_test, y_test*) data.



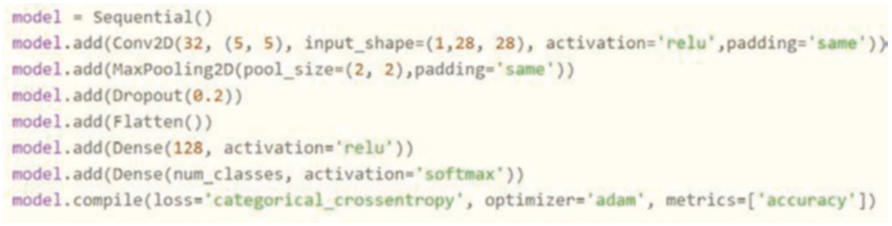
```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

Preprocessing the training and testing images.



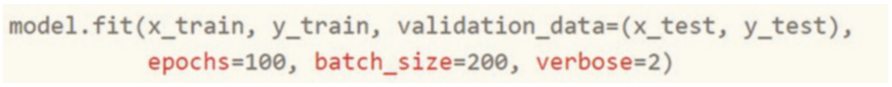
```
x_train = x_train.reshape(x_train.shape[0], 1, 28, 28).astype('float32')
x_test = x_test.reshape(x_test.shape[0], 1, 28, 28).astype('float32')
x_train = x_train / 255
x_test = x_test / 255
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Defined and compiled CNN model.

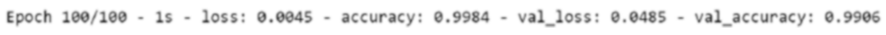


```
model = Sequential()
model.add(Conv2D(32, (5, 5), input_shape=(1,28, 28), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

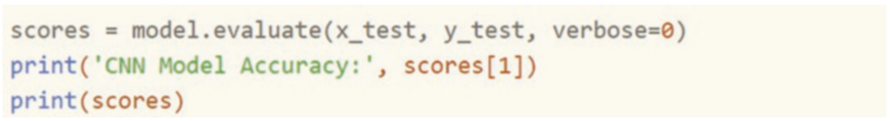
The training and validation data is used to train the CNN model.



```
model.fit(x_train, y_train, validation_data=(x_test, y_test),
          epochs=100, batch_size=200, verbose=2)
```



```
Epoch 100/100 - 1s - loss: 0.0045 - accuracy: 0.9984 - val_loss: 0.0485 - val_accuracy: 0.9906
```



```
scores = model.evaluate(x_test, y_test, verbose=0)
print('CNN Model Accuracy:', scores[1])
print(scores)
```

```
CNN Model Accuracy: 0.9905999898910522
[0.048527146894724975, 0.9905999898910522]
```

References

1. Keras, “Developer Guide,” *Google*, 2020. [Online]. Available: https://keras.io/guides/
2. F. J. John Joseph, “Working with Keras,” *GitHub*, 2020. [Online]. Available: https://github.com/ferdinjoe/Chapter-4-Working-with-Keras
3. Belsley, D.A., Kuh, E., Welsch, R.E.: Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, vol. 571. Wiley (2005)
4. Joe, J.F., Ravi, T., Justus, C.J.: Classification of correlated subspaces using HoVer representation of Census Data. In: 2011 International Conference on Emerging Trends in Electrical and Computer Technology, pp. 906–911 (2011)
5. Golle, P.: Machine learning attacks against the Asirra CAPTCHA. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 535–542 (2008)
6. Chatterjee, S., Bhoumik, S., Sarkar, A., Kumar, A., John Joseph, F.J.: Covid 19 prediction from X ray images using fully connected neural network. In: 11th International Conference on Computational Systems – Biology and Bioinformatics (2020)
7. Deng, L.: The mnist database of handwritten digit images for machine learning research [best of the web]. IEEE Signal Process. Mag. **29**(6), 141–142 (2012)

Deploying Deep Learning Models for Various Real-Time Applications Using Keras

D. Sumathi and Kumarraju Alluri

1 Keras

It is an enriched level API (Application Program Interface) of TensorFlow used for the construction of models, and it is found to be similar in stacking layers. At this point, Keras is also used to compile the developed models with several functions like loss and optimizer. Also, the training process is done with the fit function. The "backend engine" takes the responsibility of handling the low-level API. In this, the term "backend" refers to all low-level computations done through several libraries, namely Theano or TensorFlow. The default backend engine is the TensorFlow. Various other backend engines, such as Theano or the Microsoft Cognitive Toolkit or CNTK, also could be deployed based on the need. The features of Keras are:

- Easy to learn and use
- A multi backend which aids in the coding task to be rapid.
- Fast prototyping
- Integration with the low-level through deep learning libraries such as TensorFlow
- Processing of huge volumes of data and increase in speed for training the model

We can install Keras using the following steps:

1. The "pip installKeras" command can be used to deploy Keras in your environment.
2. We can confirm the installation of Keras with its version using "pip list | grep
3. Keras"

D. Sumathi (✉) · K. Alluri
VIT-AP University, Amaravati, Andhra Pradesh, India
e-mail: bksp.kumar@vitap.ac.in

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_5

4. CloningKeras can be completed with the git command “git clone https://github.com/kerasteam/keras.git”

2 Keras Models

The core data structure of Keras is the model. The models could be categorized as sequential, functional, shared layers, standard network, and multiple input and multiple output models.

2.1 Sequential Model

Layers are arranged to construct a model in Keras. A model could be defined as a collection of layers. The sequential model is suitable when there is an ordinary heap of layers in which there must be one input and one output tensor in each layer. The sequential model can be created and it consists of a sequence of layer functions. Artificial neural networks have layers in a linear order in which the data is transferred from one layer to another layer in a specified order until the data reaches the output layer. Let us see the steps in the construction of a sequential model. The design of the sequential model could be represented as shown in Fig. 1.

Step 1:

Sequential API is used to create the model. The required Keras classes must be imported. In this case, the model is sequential. Layers required for the model construction are dense and activation.

Step 2:

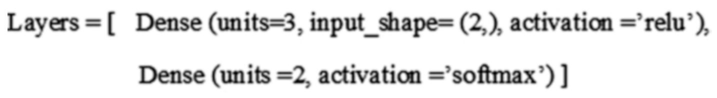
An instance for the sequential model could be created, and it is assigned to the variable model.

model = Sequential(layers)

Step 3:

We need to define the layers. There are various types of layers. Here we have used the dense layers. For example, consider the neural network that has been depicted in Fig. 2. Here, there are five neurons in the input layer. The hidden layer comprises four neurons in the hidden layer and two neurons in the output layer.

Input layer is the first layer, hidden layer acts as the second layer, and the output layer is the third layer. Hidden layer nodes are linked with the nodes in the output layer, and hence it is a densely connected layer:



```
Layers = [   Dense (units=3, input_shape= (2,), activation ='relu'),
             Dense (units =2, activation ='softmax') ]
```

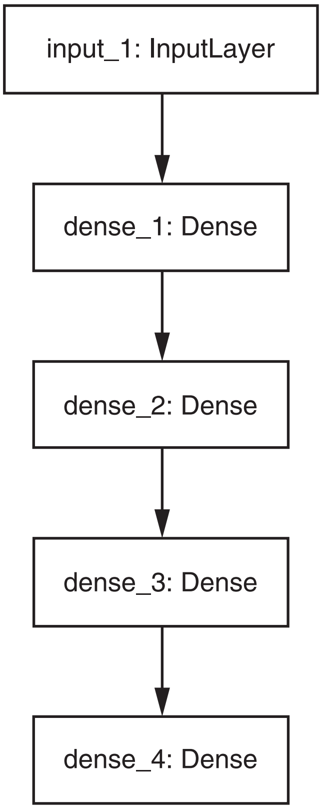


Fig. 1 Sequential model architecture

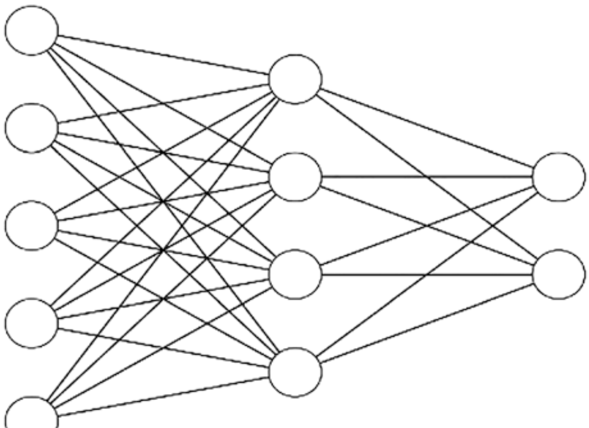


Fig. 2 Architecture of a sample ANN

In the above example, the input in the input layer is 2 and it is represented by the argument input_shape. The activation functions that are used over here are softmax and rectified linear unit (ReLu).

Apart from the dense layers, other layers such as pooling, recurrent, and convolutional could be utilized based on the application.

The convolutional layers could be used for the construction of the models that deal with the images, and the recurrent layers are used in the models that work with the time-series data.

Step 3:

The other way of creating the input layer and the hidden layers in the sequential model could be given as:

```
model.add (Dense (units=10, input_shape =2, activation =’relu’, kernel_initializer=’uniform’))
```

The arguments that are passed to the function are given as below:

Add function is used to deploy in all layers.
Dense() represents that the neural network is densely connected
input_shape and units are used to denote the number of nodes in the input layer and the hidden layer, respectively.
The activation function is applied to each node. Sigmoid, linear, and ReLu are various activation functions.
Kernel_initializer is used to assign the preliminary random weights of the layer.

Step 4:

The hidden and output layers are constructed with the following functions, respectively:

```
model.add(Dense(units=15, activation =’relu’,kernel_initializer=’unitform’))

model.add(Dense(units=1, activation =’sigmoid’,kernel_initializer=’unitform’))
```

The activation functions used over here could be classified as sigmoid, rectified linear unit, and softmax.

The ANN classifier has to be compiled. The syntax is:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Before training, the model has to be compiled with optimization and loss function. There are various optimization functions such as SGD, Nadam and Adam.

Based on the application, one can choose the optimizer. The loss function is used to compute the errors and losses. The nature of the application decides the loss function. The last argument ‘accuracy’ measures the accuracy of the model.

Step 5: The model has to fit with the training datasets. To fit the model, the period is set to 20 and the group size is set to 20 using the syntax model.fit().

In this, the epochs refer to a single execution through all the data and the batch size denotes the number of samples that could be given for execution at a time.

Step 6: Evaluation of the model must be done on the test set with evaluate ().

2.2 Keras Functional Model

To solve complex models, functional API could be implemented. When the problem comprises of huge dimensions or complexities, then it is recommended to deploy the functional model. When there are multiple inputs and outputs, a model could be designed with the shared layers, which make use of the functional API. The first step is to import the models and layers through the import function.

For example, assume the shape of the data is a group of 6000-dimensional vector. In this case, only the shape of each sample is taken for the development of the model.

```
inputs = keras.Input(shape=(6000,))
```

The first phase is to specify the input. Here assume, the tensor shape of (32,32,3) is considered as the sample of the shape.

Therefore, the input layer has to be created with the following syntax:

```
input_layer = keras.Input(shape=(32, 32, 3))
```

The next layer is the connecting layer. Here, a pairwise connection is performed by denoting the origin of input while identifying each new layer.

To the dense layer function, units and activation functions are used. Here ‘ReLu’ activation function is used:

```
dense = layers.Dense(64, activation='relu')
x = dense(input_layer)
print(x)
```

To add more layers, the dense function is called with 32 units and ReLu activation function is used:

```
x = layers.Dense(32, activation='relu')(x)

outputs = layers.Dense(12)(x)

print(outputs)
```

In the output layer, 12 neurons are assigned, and the activation function assigned here is softmax. At last, the Keras model is constructed with the inputs and outputs. A model is created with the specified inputs and outputs with the following syntax:

```
model = keras.Model(inputs=inputs, outputs=outputs, name='model')
```

model.summary() provides the details of the functional model.

```
Model: "model"
_________________________________________________________________
Layer (type)             Output Shape            Param #
=================================================================
input_24 (InputLayer)       [(None, 32, 32, 3)]      0
_________________________________________________________________
dense_28 (Dense)          (None, 32, 32, 64)      256
_________________________________________________________________
dense_29 (Dense)          (None, 32, 32, 32)      2080
_________________________________________________________________
dense_30 (Dense)          (None, 32, 32, 12)      396

Total params: 2,732
Trainable params: 2,732
Non-trainable params: 0
```

2.3 *Standard Network Models*

Few standard network models such as multilayer perceptron(MLP), convolutional neural network (CNN), and recurrent neural network(RNN) have been described in detail.

2.3.1 Multilayer Perceptron (MLP)

It is one of the conventional types of neural networks. An MLP is defined as a perceptron in which groups with several perceptrons are arranged in layers to resolve complex issues. Figure 3 shows an MLP with three layers in which the first layer is the input layer, which transmits the signal to the second layer, known as the hidden layer, which in turn passes the signal to the final layer (output layer).

The main application of MLP is to predict the classifications based on the label or class. It is mainly appropriate for the real-valued numbers, which are considered as the input. Among the various inputs, the output is computed. A combination is done based on the input weights, and then it is given to the output with the help of some nonlinear activation function. It could be defined as in Eq. 1:

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b = \varphi\left(W^T x + b\right) \tag{1}$$

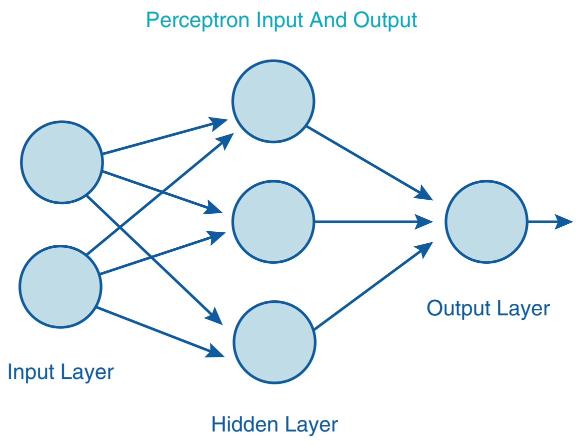


Fig. 3 Architecture of MLP

In Eq. 1, w represents the weights of the neurons, x as the inputs, b refers to the bias function, and the activation function is denoted as φ.

Neurons that are deployed in this network are used to acquire knowledge about the non-linear representations. The activation function is used to familiarize the non-linearity into the output of a neuron. Several activations could be deployed based on the applications.

Sigmoid Function:

The sigmoid function is a special case of logistic function which is expressed as given in Eq. 2:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2}$$

It is mainly used to control the sum of data inputs once the weights are applied. The output after the application of the function lies in the interval of 0 to 1.

Tanh Function:

During the deployment of sigmoid functions, there is a likelihood of getting stuck in the mid of the function. The reason behind this struck is due to the presence of negative input. This drives the output to zero. Feedforward activations are used to compute the parameter gradients which leads to less frequency in updating the model attributes. Hence, the break in the function occurs. An alternate approach to sigmoid is the usage of the tanh function.

The tanh function could be defined as:

$$\tanh(x) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3}$$

This is also similar to the sigmoidal function but the value ranges from -1 to +1.

Softmax Function:

This function is used to identify the possibilities of multiple classes. The output of the function is deduced in terms of the probability. It is used in the output layer. The constraint in this function is that if the classes grow, then the cost of deploying the softmax function would be high:

$$s(x) = \frac{e^{x_i}}{\sum_j e^{x}{}_j} \tag{4}$$

The output of this function will be between 0 and 1.

Rectified Linear Unit (ReLu) function:

This function generates the output if the input is positive, else it will give the output as zero. Many neural networks work based on this activation function, which is treated as a default due to the better performance of the model, and training the model is easier:

$$\mathrm{ReLu}(\mathrm{x}) = \max(0,\mathrm{x}) \tag{5}$$

2.3.2 Convolutional Neural Network (CNN)

CNN architecture comprises of three parts. They are described as given below:

Convolutional layer: The objective of this layer is to mine the features from the input source image. The spatial association among the pixels is maintained by acquiring the image features with the help of small squares of input data. Image size is reduced since it might be easy to process the data without dropping the features which are found to be perilous to get a good prediction. During this function, a kernel or filter or feature detector is used. The feature map or activation map is obtained from this layer. When the values of filter are varied, then different feature maps or activation maps would be obtained for the same input image. Recognition of patterns is done in a better way by running more number of filters, which in turn results in the extraction of more features. At the end of every convolution operation, non-linear functions must be applied.

Pooling layer: Downsampling of features is done in this layer. This is done so that the dimensionality is decreased. The functionalities of pooling layer are:

- Dimensions of the feature is reduced.
- The count of constraints and calculations in the network are decreased.
- There will be no change in the network for small alterations and transformations in the input image for maximum value in a local neighborhood.

Fully connected layer: Features determined in the previous layers are levelled into the vectors, and the likelihood of the image is done. The goal of this layer is to classify the input image based on the features into several classes. Classification depends on the training dataset. The architecture of CNN is depicted in Fig. 4.

In Keras, the following steps are adapted to build the model:

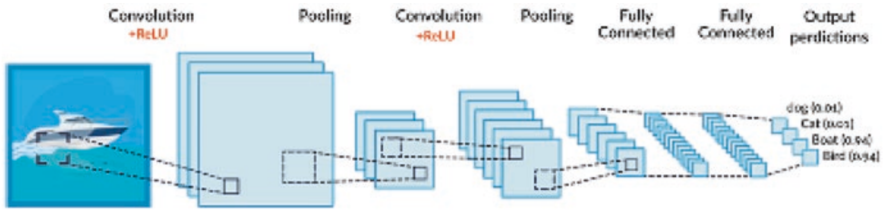


Fig. 4 Architecture of CNN

1. The input data has to be reshaped into the format which is appropriate for the convolutional layers with the help of reshape() function.
2. One hot encoding is used to translate the classes by using to_categorical() function in class-based classification.
3. The model has to be constructed with the help of sequential.add(). To construct a 2D convolutional layer,

```
model.add(Conv2D(64, kernel_size=(7,7), activation='relu', input_shape(28,28,1))
```

4. The pooling layer is added as the next phase.
5. Flatten layers must be created to generate the vector for the fully connected layers.
6. If required, add one or more fully connected layers.
7. The model must be compiled.
8. Train the model and then predict the class.

2.3.3 Recurrent Neural Networks (RNN)

It belongs to a category of neural network. It works well for the construction of the model which could deal with time-series data or in natural language processing (NLP). Several RNN architectures exist. Number of inputs and outputs is used for categorization. They are many to many, many to one and one to many. Many to many are used for translation purposes. Many to one are applied for sentiment analysis and one to many could be applied for the generation of sequence.

2.4 Shared Layers Model

From the name, it could be interpreted as there might be multiple layers that are utilized to infer the output from a feature extraction layer or several feature extraction layers from an input.

In the shared input layer, several convolutional layers with various sizes of kernels are used to deduce the input image. The output of a Long Short-Term Memory (LSTM) feature extractor for the sequence classification could be determined by applying submodels in a parallel manner.

2.5 Multiple Input and Output Models

The objective of these models is to build a complex model in which it takes more than one input and output.

In the case of multiple inputs, more than one input would be given. For example, the main input layer and the auxiliary input layer are provided as given below:

```
one_inputs =Input(shape= (32,1))
in_layer=Conv1D (16,5, activation='ReLu') (one_inputs)
in_layer=AveragePooling1D (3) (in_layer)
in_layer = Flatten () (in_layer)
```

The sequencing of data is done with the help of Conv1D. The output feature map is constructed through this Conv1D layer. The input is passed over the filter and the values are multiplied element-wise so that the output feature map is created. Next, the values are flattened into a single dimension:

```
a_input=Input(shape=(12,1))
in_layer=Concatenate ()([in_layer,a_input])
```

The auxiliary input is added in the middle as an additional input to the final dense layer. In this example, the input is one dimension. Then both the input layers are concatenated and processed through the dense layer:

```
Outputs=Dense(20,activation='sigmoid')(in_layer)
New_model = Model(inputs=[one_inputs,a_input], outputs=outputs)
```

In the same way, multiple output models could be developed.

3 Comparison of Frameworks

A comparison of TensorFlow, Keras, and PyTorch is given in Table 1.

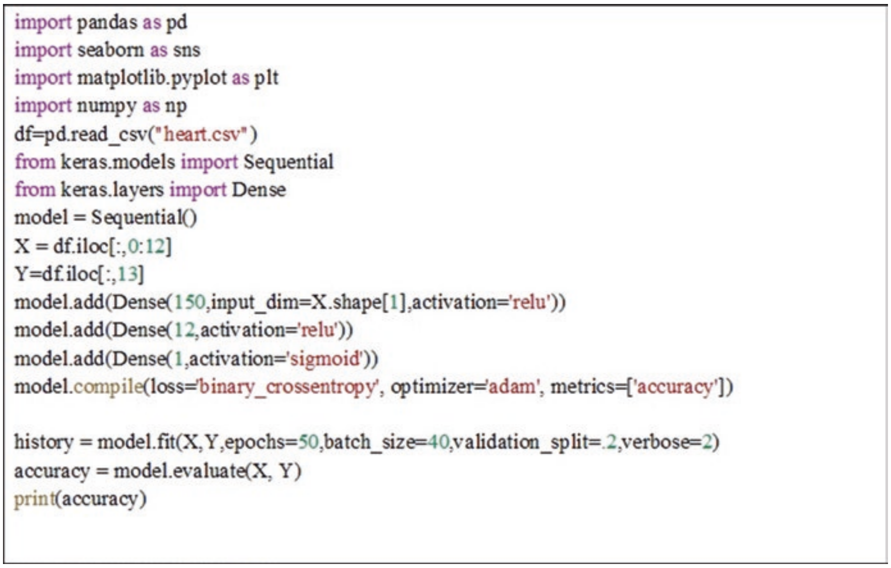
4 An Illustration of the Sequential Model

For instance, sequential model construction is illustrated.

The following is an example for sequential model construction:

Table 1 Keras vs TensorFlow vs PyTorch

Features	Keras	TensorFlow	PyTorch
Supported languages	It supports Python language.	It supports Go, Python, C++,and Javascript.	Supported by Java, C++, and Python.
API	High-level API could be experienced.	Both high-level and low-level API could be provided.	Low-level API.
Architecture	Simple.	Not easy to use.	Complex.
Datasets	Used for small datasets and the performance of the model is slow.	Deployed for large datasets and the performance of the model is high.	
Models	Neural network models could be built.	Various computational methods are used.	Models make use of NLP and neural networks.



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
df=pd.read_csv("heart.csv")
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
X = df.iloc[:,0:12]
Y=df.iloc[:,13]
model.add(Dense(150,input_dim=X.shape[1],activation='relu'))
model.add(Dense(12,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

history = model.fit(X,Y,epochs=50,batch_size=40,validation_split=.2,verbose=2)
accuracy = model.evaluate(X, Y)
print(accuracy)
```

Output:

```
303/303 [==============================] - 0s 36us/step
[0.7296317540379641, 0.5445544719696045]
```

From the output, the loss obtained is 72% and accuracy is 54%. From this, it is understood that the hyper parameters must be tuned to get the precision for the model.

5 Unstructured data and Structured Data

5.1 *Unstructured Data*

Unstructured data might be in terms of service logs, social media data, and complaints. Extensive research is carried out in analyzing the impact on the kinds of data on business consequences. Apart from that, online reviews on services and products are also focussed. Most of the research works aim at determining the impact of reviews and providing mechanisms to include the factors in a predictive model. A detailed analysis of the usage of unstructured data by various researchers has been presented in Table 2.

The companies intend to improve the performance of sales. This could be done through various social media sites. Through these reviews, responses are recorded and a detailed analysis of the current datasets must be done with the help of several predictive models. This helps a company to attract more users, improve the products, and modify its service based on customer needs. A machine learning model could be developed to determine the value and relationships between the different kinds of data produced by the content.

Case Study 1:
Simplified text processing is to be done for textual data. A simple API could be provided to perform natural language processing (NLP) actions like sentiment analysis, translation, classification, extraction of a noun phrase, and part-of-speech tagging. In this, the sentiments of tweets that are posted on twitter are analyzed for its polarity.

Table 2 Analysis of researchers' work on unstructured data

Sl. no	References	Implementation.
1	Sun et al. [1]	Forecasting of sales is done with the help of extreme learning methods (ELM).
2	[2] Thomassey and Happiette	Prediction of sales is done by the implementation of soft computing methods such as fuzzy inference systems and neural networks.
3	Teucke et al. [3]	Decision trees are applied to identify articles that need to be re-arranged. Then support vector machines are applied to determine the original forecasts to achieve accuracy.
4	[4], Yu et al	An analysis of online movie reviews dataset is done in such a way that the sentiments are expressed. Detection of the sentiments is done through probabilistic latent semantic analysis (PLSA). Sales prediction is done through the deployment of an autoregressive sentiment-aware model.
5	[5–9]	An extensive study has been done on blogs, news, and social network services (SNS) to perform the correlation analysis among the public emotions and stock prices.
6	[10]	Support vector machine has been deployed to retrieve the information from news articles to forecast the intraday price movements of economic assets.

Step 1:

TextBlob is a library used for the processing of the text. Import it. Take a tweet and assign it to the text.

```
from textblob import TextBlob

text = '''

"'Congratulations to Rob Jones in accounting for winning our #NFL football
pool!'; with a picture of a victorious Rob.

'#TGIF, we've been working like a dog!'; with a photo of a sleeping puppy.

Did you see Peyton Manning's twisted ankle? Should've been wearing our
#AcmeOrthotics; with a link to the product.
```

Step 2: An object for the TextBlob is created.

Step 3: Tokenization could be done by retrieving the tokens by accessing the properties of blob objects such as sentences and words.

```
blob = TextBlob(text)

blob.tags

[('"', 'JJ'), ('`', 'JJ'), ('Congratulations', 'NNS'), ('to',
'TO'),  ('Rob',  'NNP'),  ('Jones',  'NNP'),  ('in',  'IN'),
('accounting', 'VBG'), ('for', 'IN'), ('winning', 'VBG'),

blob.noun_phrases

WordList(['" `', 'congratulations', 'rob jones', 'nfl', 'foot-
ball pool', 'rob', '` #', 'tgif', '' ve', 'peyton manning',
'' s', '' ve', 'acmeorthotics'])

for sentence in blob.sentences:

print(sentence,sentence.sentiment.polarity)

"'Congratulations to Rob Jones in accounting for winning our
#NFL football pool!............. 0.625
'#TGIF, we've been working like a dog!'………. 0.0
Did you see Peyton Manning's twisted ankle? -0.5
Should've been wearing our #AcmeOrthotics; ………….. 0.0
```

From the output, it is understood that the polarity of sentences are computed individually and displayed.

5.2 Structured Data

Structured data is in the form of tables. Examples of structured data are in the format of CSV, XLS, and TXT files. In these files, a delimiter is used to separate the variables. The delimiter is either a variable or fixed width. Data must be analyzed and needs a clear understanding. For further analysis, it must be pre-processed, analyzed, examined, cleaned, and visualized. Data needs an extensive exploration so that a clear understanding could be gained. Moreover, one can determine evidence from the data, do formulation of assumptions and hypothesis, and perform verification of the quality of data for further analysis.

Case Study: Keras on Structured Data
Classification of structured data could be done. Keras could be used to deploy the model. Let us see the behavior of the data. The dataset taken for analysis is: Real estate.csv

To start with,

Step 1: Import the required libraries such as pandas, TensorFlow, and NumPy. Pandas is used to read the data and store it in the data frame.

Step 2: Load the data.

```
dataset =pd.read_csv('Real estate.csv')

# for reading and importing the dataset

# data is sourced from Kaggle
```

Step 3: Explore the data.

```
dataset.columns
```

```
Index(['No', 'X1 transaction date', 'X2 house age',

       'X3 distance to the nearest MRT station',

       'X4 number of convenience stores', 'X5 latitude', 'X6 longitude',

       'Y house price of unit area'],
```

```
dtype='object')
```

Columns present in the datasets are transaction dates related to the house, age of the house, distance from the house to the nearest metro, the number of stores near to the house, geographical coordinates, and the house price.

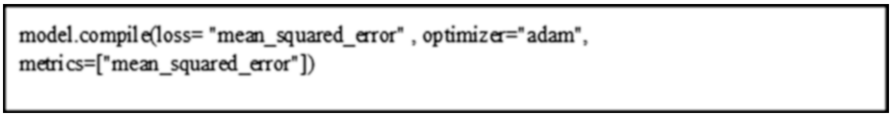
Step 3: A deep learning model could be built by defining three layers, namely input, hidden, and the output layer.

The first layer is defined as the input layer with dimensions that have to be set for the independent variables. Here, the number of independent variables is six. The hidden layer and the output layer are defined with the neurons. In the output layer, the number of neurons is one to predict the price of the house.

```
from keras.models import Sequential

from keras.layers import Dense

model = Sequential()

model.add(Dense(50, input_dim=6, activation= "ReLu"))

model.add(Dense(20, activation= "ReLu"))

model.add(Dense(1))

model.summary() #Print model Summary
```

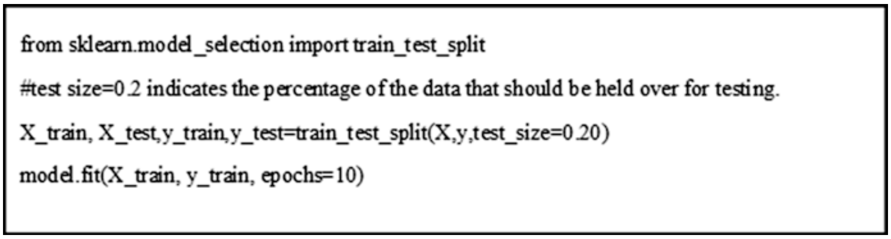
Input layer neurons = 6. First hidden layer neurons =50. Second hidden layer neurons = 20. Output layer = 1 neuron.

Step 4: The model has to be compiled. The weights and biases could be changed with the optimizer. The performance of the model could be evaluated with the loss metric and mean squared error.



```
model.compile(loss= "mean_squared_error" , optimizer="adam",
metrics=["mean_squared_error"])
```

Step 5: The model has to fit on the training set. The training set and the test set are taken in the ratio of 80:20.



```
from sklearn.model_selection import train_test_split

#test size=0.2 indicates the percentage of the data that should be held over for testing.

X_train, X_test,y_train,y_test=train_test_split(X,y,test_size=0.20)

model.fit(X_train, y_train, epochs=10)
```

The number of epochs denotes the frequency of the model to run. There is an association between the performance of the model and the epochs. If more epochs are run, then there will be an improvement in the model.

```
Epoch 1/10 331/331 [==============================] - 0s 69us/
step - loss: 34.2135 - mean_squared_error: 34.2135

Epoch 2/10 331/331 [==============================] - 0s 57us/
step - loss: 33.0375 - mean_squared_error: 33.0375

Epoch 3/10 331/331 [==============================] - 0s 51us/
step - loss: 32.1954 - mean_squared_error: 32.1954

Epoch 4/10 331/331 [==============================] - 0s 42us/
step - loss: 31.2572 - mean_squared_error: 31.2572

Epoch 5/10 331/331 [==============================] - 0s 42us/
step - loss: 30.4676 - mean_squared_error: 30.4676

Epoch 6/10 331/331 [==============================] - 0s 42us/
step - loss: 29.7005 - mean_squared_error: 29.7005

Epoch 7/10 331/331 [==============================] - 0s 47us/
step - loss: 29.0260 - mean_squared_error: 29.0260

Epoch 8/10 331/331 [==============================] - 0s 48us/
step - loss: 28.1258 - mean_squared_error: 28.1258

Epoch 9/10 331/331 [==============================] - 0s 47us/
step - loss: 27.4899 - mean_squared_error: 27.4899

Epoch 10/10 331/331 [==============================] - 0s 63us/
step - loss: 26.4776 - mean_squared_error: 26.4776
```

If the mean squared error value is low, then the model performs better. From the output, it is inferred that the mean squared error value is less.

6 Deploying Deep Learning Workstation

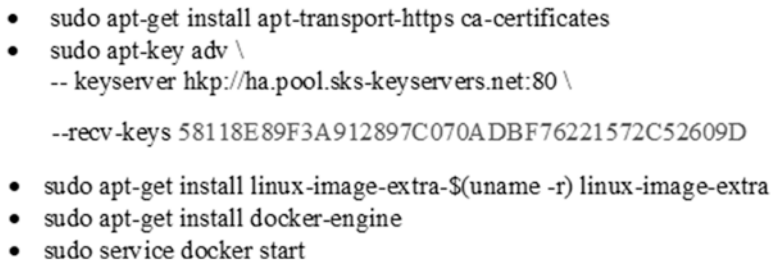
The hardware requirements for building deep learning workstation vary based on the situation or scope of work under consideration. Irrespective of it, for the reasonable performance of the system, the minimum requirements are RAM-32 GB, GPU- NVIDIA, and hard disk of 1 TB. The tricky phase is to select the right GPU, and it depends on several factors, such as: How many tensor cores are available? How many FLOPs are given in the GPU? Is memory bandwidth high? Does GPU have a minimum of 16-bit processing capabilities?

Before one chooses to build their deep learning workstation, one has to know the risks involved in doing that, such as the system can become obsolete after some time and there is a chance of additional cost incurred due to hardware failures/maintenance. If your business requirements do not overload at a high pace and you are worried about the security, then it is a better option for personal deep learning workstation.

Once the GPU, RAM, and hard disk are assembled to the motherboard, then follow the below steps:

1. Install the operating system (OS): Any OS is okay, but an open source server version is highly preferred (here, we have chosen Ubuntu Server).
2. Install Python and R libraries.
3. Install drivers of your GPU (here, NVIDIA):

   sudo apt-get install nvidia-367
4. Create a virtual container to use the services dynamically:



   - sudo apt-get install apt-transport-https ca-certificates
   - sudo apt-key adv \
     -- keyserver hkp://ha.pool.sks-keyservers.net:80 \
     --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
   - sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra
   - sudo apt-get install docker-engine
   - sudo service docker start

   - You can create a separate user for using Docker Engine and run its services.
5. Integrate Docker services with GPU.
6. Install DL libraries in the Docker:
   - git clone https://github.com/saiprashanths/dl-docker.git
   - cd dl-docker
   - docker build -t floydhub/dl-docker:gpu -f Dockerfile.gpu

You are ready for running deep learning algorithms in your workstation.

7 Binary Classification

Situation Description

Assume that you are the owner of an online music store and you release the music before they release into the market as a premium offer. On a few occasions, you observed some customers are using this service and some or not. As an owner, you are interested in the customers who return, based on which you can have target advertising and discounts. To solve this problem, you approached a local startup and they solved the problem in the following way:

Steps:

1.	Load the dataset with the read_csv function. The dataset used here is songs_review_with_column_names.csv
2.	Understand the data df.dtypes
	```
	ID                 int64
	Album_length       float64
	Price              float64
	Review_1           int64
	Review_2           float64
	Minutes_listened   float64
	Completion         float64
	Generated_requests int64
	Target             int64
	dtype: object
	```
	All the datatypes of the columns are correctly structured and there are no issues. Let me explain the attributes present in the dataset: ID is the customer ID, Album_length indicates the length album in minutes, price tells the cost in dollars, and Review_1 is given by the user (0- not liked, 1-liked). Review_2 is given by the critique in the range of 1 to 10. Minutes_listened tells you the number of minutes actually the song/album was heard. Completion indicates whether the customer completed the listening of the album, Generated_requests indicates the number of times the customer took the help of customer service to use the site, and finally, ‘Target’ is the variable which indicates the likelihood of the customer to returning to the store, i.e. 1-return and 0-not returned.

3. Shuffle the data as most of the initial rows belong to the customers who are likely to return, and this will create inefficiency during the testing phase.

```
df = df.sample(frac=1).reset_index(drop=True)
```

4 Checking for class imbalance.

```
df.Target.value_counts()
```

```
#O/P:
0    11847
1     2237
Name: Target, dtype: int64
```

Here, not likely to visit back are very high. This would lead the system to predict the majority class and ignore or consider the minor class as noise. To avoid these sorts of problems, we will balance it irrespective of data loss.

#Balancing the dataset.

```
import NumPy as np

indexNames = df[df['Target'] == 0].index

break_point = (df.shape[0]-int(np.sum(df.Target)))-int(np.sum(df.Target))

count = 0

for i in indexNames:

    count = count + 1

    df.drop(i,inplace=True)

if(count==break_point):        break
```

```
#confirming class balance
df.Target.value_counts()
1    2237
0    2237
Name: Target, dtype: int64
```

The dataset is perfectly balanced. In real-time, you no need to absolutely balance but we should make sure that class imbalance should not lead to misleading conclusions.

5. Dividing the data into training, validation, and testing.

```
from sklearn.model_selection import train_test_split

df_train, df_val, df_test = split_stratified_into_train_val_test(df, stratify_colname='Target',

frac_train=0.80, frac_val=0.10, frac_test=0.10)
```

Of the actual data, 80 % is considered for the training, 10 % for validation and 10% for testing the model.

We divide each subset of data (training, validation, and testing) into inputs and targets.

```
train_inputs = df_train.iloc[:,1:8]

train_targets = df_train.iloc[:,8]

val_inputs = df_val.iloc[:,1:8]

val_targets = df_val.iloc[:,8]

val_inputs = df_val.iloc[:,1:8]

val_targets = df_val.iloc[:,8]
```

6. Normalizing the inputs.

```
min-max = preprocessing.MinMaxScaler()

data_train= min-max.fit_transform(train_inputs)

changed_train_inputs = pd.DataFrame(data_train)
```

Here, we have normalized the inputs of the training set. We can follow the same process for validation and test datasets.

7. Saving the preprocessed dataset into .npz format.

```
np.savez('data_train', inputs=changed_train_inputs, targets=train_targets)

np.savez('data_validation', inputs=changed_val_inputs, targets=val_targets)

np.savez('data_test', inputs=changed_test_inputs, targets=test_targets)
```

8. Load the processed datasets.

```
import NumPy as np

import tensorflow as tf

npz = np.load('data_train.npz')

train_inputs = npz['inputs'].astype(np.float)

train_targets = npz['targets'].astype(np.int)

npz = np.load('data_validation.npz')validation_inputs, validation_targets =
npz['inputs'].astype(np.float), npz['targets'].astype(np.int)

npz = np.load('data_test.npz')

test_inputs, test_targets = npz['inputs'].astype(np.float), npz['targets'].astype(np.int)
```

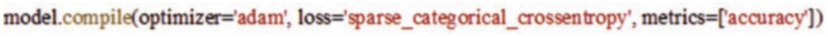
9.

```
Import NumPy as np

import tensorflow as tf

input_size = 7

output_size = 2

hidden_layer_size = 100
```

Setting the parameters for the neural networks model. In our dataset, we have seven input variables (i.e., input_size=7) and a class label with two categories (i.e., output_size=2). Building the model by specifying the required parameters. Keras layers have various properties regarding the customization of the neural network architectures. We have used Dense layers in which we have set the activation function and size of hidden layers. You can experiment by further changing the number of hidden layers, the size of the hidden layers, changing the activation function, etc.

```
model = tf.keras.Sequential([

    tf.keras.layers.Dense(hidden_layer_size, activation='ReLu'),

    tf.keras.layers.Dense(hidden_layer_size, activation='ReLu'),

    tf.keras.layers.Dense(output_size, activation='softmax') ])
```

The most commonly used adam optimizer is to compile the Keras model with loss and accuracy being reported at each epoch.



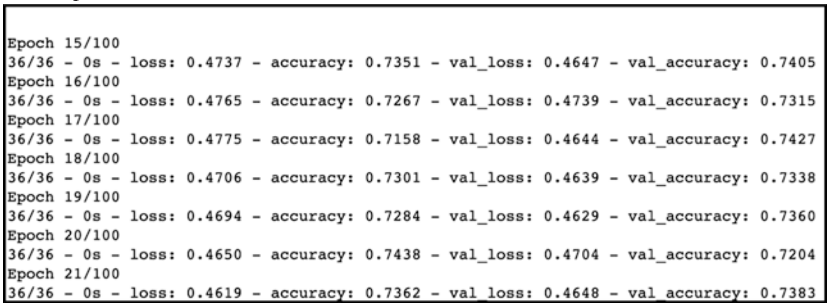
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Setting other relevant parameters to finally build the model.

```
batch_size = 100

max_epochs = 100

early_stopping = tf.keras.callbacks.EarlyStopping(patience=3)

model.fit(train_inputs,

          train_targets,

          batch_size=batch_size,

          epochs=max_epochs,

          callbacks=[early_stopping],

          validation_data=(validation_inputs,

          validation_targets),

          verbose = 2
```

model.fit() function would facilitate the data analyst with various options. For example, to avoid overfitting of the model, we have used early stopping. This would help in automatically stopping the model building where your loss increased when compared with the previous iteration. Sometimes, loss increases would be very small, and this should not stop the model building. This is handled by setting another parameter called patience (allows the loss to increase till specified number of times).

Since verbose is set to true, we could see the results (accuracy vs loss) for every epoch, and the sample of the same is shown below:



```
Epoch 15/100
36/36 - 0s - loss: 0.4737 - accuracy: 0.7351 - val_loss: 0.4647 - val_accuracy: 0.7405
Epoch 16/100
36/36 - 0s - loss: 0.4765 - accuracy: 0.7267 - val_loss: 0.4739 - val_accuracy: 0.7315
Epoch 17/100
36/36 - 0s - loss: 0.4775 - accuracy: 0.7158 - val_loss: 0.4644 - val_accuracy: 0.7427
Epoch 18/100
36/36 - 0s - loss: 0.4706 - accuracy: 0.7301 - val_loss: 0.4639 - val_accuracy: 0.7338
Epoch 19/100
36/36 - 0s - loss: 0.4694 - accuracy: 0.7284 - val_loss: 0.4629 - val_accuracy: 0.7360
Epoch 20/100
36/36 - 0s - loss: 0.4650 - accuracy: 0.7438 - val_loss: 0.4704 - val_accuracy: 0.7204
Epoch 21/100
36/36 - 0s - loss: 0.4619 - accuracy: 0.7362 - val_loss: 0.4648 - val_accuracy: 0.7383
```

10. Applying the model on the test data.

```
test_loss, test_accuracy = model.evaluate(test_inputs, test_targets)
```

O/P:

```
14/14 [==============================] - 0s 1ms/step - loss: 0.4663 - accuracy: 0.7634
```

Overall accuracy of 76.34 is observed.

Conclusion:
The owner of the music store could predict about customers visiting back or not with 77 % accuracy. The performance would vary based on the hyperparameters and the activation functions.

8 Multiclass Classification

Scenario Description

Handwritten digits' classification is a classical problem of deep learning. In the literature, researchers around the world used this dataset to test their proposed solutions. In this process, we have seen "DropConnect based regularization" achieve the highest accuracy, above 99.5 %, and one of the lowest accuracies of 85.47 % was observed with target coding of the deep neural networks. Other relevant works on the same dataset can be seen in [11–14] achieved good performance, but all of them customized the underlying deep neural network architectures. In this chapter, we could achieve an accuracy of 98.5 % without using CNN and following the basic image-flattening process.

In the MNIST (Modified National Institute of Standards and Technology) site, following the traditional process, they achieved an accuracy of only 92 % and we aim to increase this without touching the advanced neural network architectures. This chapter aims to give a sense of accuracy improvement by fiddling the hyperparameters.

About the dataset: MNIST has 70,000 handwritten digits, and each time the neural network is fed with the handwritten image as input and predicts the digit among 10 possibilities. Each image is a 28×28 pixel grey-scaled image. The pixels range from 0 to 255, where 0 indicates purely black and 255 refers to pure white. As our focus is not to use CNN for this, we will flatten each image by which we will have 784×1 vector as input to the neural network. To add the non-linearity, we can add the hidden layers based on our requirement and the intermediate results are sent to the output layer with ten neurons.

The broad steps that we follow are:

A. Load and preprocess.
B. Build a model with appropriate activation functions, loss function, and optimizers.
C. Run the model on the test data.

1. Libraries such as NumPy and tensorflow is imported.

   The dataset comes with Tensorflow datasets and we are loading it as below:

   ```
   digits_dataset, digits_info = tfds.load(name='digits', with_info=True, as_supervised=True)
   ```

   as_supervised = TRUE would load the data in two separate entities, i.e. input and target, and with_info would give metadata about the dataset which we will use in the next few lines of the code.

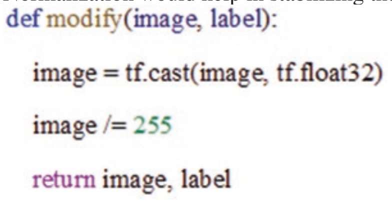
2. Partition the data into training and testing.

   ```
   digits_train, digits_test = digits_dataset['train'], digits_dataset['test']
   ```

   From the training data, we extract the validation data. We have used 10 % of training as validation.

   ```
   total_validation_samples = 0.1 * digits_info.splits['train'].total_examples

   total_test_samples = tf.cast(total_test_samples, tf.int64)
   ```

3. Normalization would help in stabilizing the results for which the following code is used:



```
def modify(image, label):

    image = tf.cast(image, tf.float32)

    image /= 255

    return image, label
```

Calling the function to apply to the training data.

```
changed_train_and_validation_data = digits_train.map(modify)

Similarly for testing data,

test_data = digits_test.map(modify)
```

4. Shuffling the training dataset.

```
BUFFER_SIZE = 10000

shuffled_train_and_validation_data                                      =

changed_train_and_validation_data.shuffle(BUFFER_SIZE)
```

We cannot shuffle the entire dataset and this is especially true with big datasets. To handle this, we specify the extent of shuffling in every instance.
Then, we divide the training data into validation data, and the remaining is treated as training instances.

```
validation_data = shuffled_train_and_validation_data.take(total_validation_samples)

train_data = shuffled_train_and_validation_data.skip(total_validation_samples)
```

5. Creating batches:

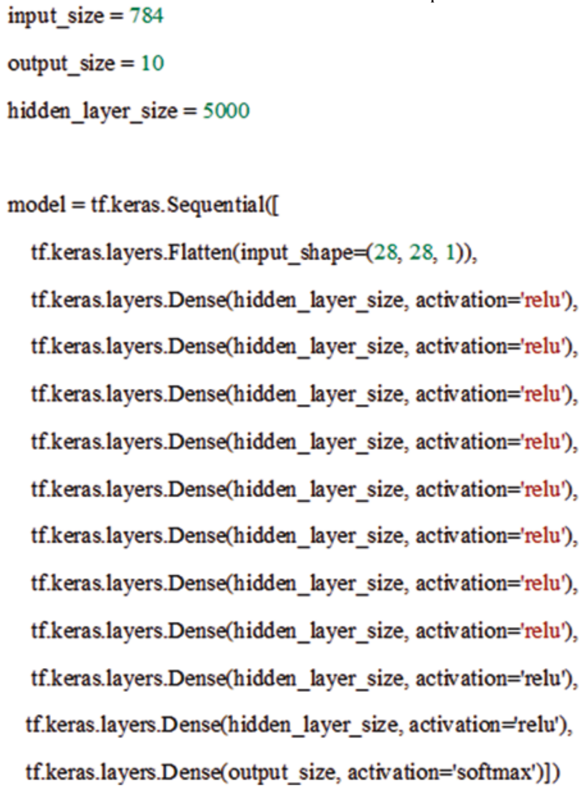
```
BATCH_SIZE = 150

train_data = train_data.batch(BATCH_SIZE)
```

Batches would help in stabilizing the results, and the size of the batch depends on the underlying dataset. We don't need to create batches for the validation data as weights will not get updated from each batch to the other in the validation set. The same concept applies to the test data. But, it is required to denote the total number of samples in validation pertains to one batch and total instances in a test as one more batch.

```
validation_data = validation_data.batch(total_validation_samples)

test_data = test_data.batch(total_test_samples)

Dividing the validation data into inputs and targets.

validation_inputs, validation_targets = next(iter(validation_data))
```

6 Building the model.

Setting hyperparameters for the model, such as hidden layer size, number of hidden layers, activation functions, etc. We use Keras flatten function as we aim to achieve good performance without using CNNs. For each hidden layer, Keras dense() function would perform dot products of inputs and weights. This would also support various activation functions, and we currently used ReLu, and for output layer, softmax activation is used as classification should be made in the context of probabilities.



```
input_size = 784

output_size = 10

hidden_layer_size = 5000


model = tf.keras.Sequential([

  tf.keras.layers.Flatten(input_shape=(28, 28, 1)),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

  tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

 tf.keras.layers.Dense(hidden_layer_size, activation='relu'),

 tf.keras.layers.Dense(output_size, activation='softmax')])
```

7. Choose the Optimizer and loss function.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',

metrics=['accuracy'])
```

We have chosen the most commonly used optimizer called “Adam,” and loss function is sparse_categorical_crossentropy as the outputs are treated as one-hot encoded variable by the model.

8. Running the model.

```
TOTAL_EPOCHS = 10

model.fit(train_data, epochs=TOTAL_EPOCHS,
validation_data=(validation_inputs,
validation_targets), verbose =2)
```

The neural network is built with specified parameters, and a model is constructed from the training data, and its understanding is cross-checked with validation samples.
O/P (Fig. 5):

9. Test the model.
Note that, once the model is built and applied on the test data, we should not go back and change the hyperparameters, and this leads to overfitting.

```
test_loss, test_accuracy = model.evaluate(test_data)
```

O/P:
Loss=1.44, Accuracy=98.56 %
Note that, if testing accuracy is close to validation accuracy, then it is an indicator of a non-overfitting scenario.

Conclusion
Achieving accuracy greater than 98% is good, which is made possible without using CNN and flattening the images. Even though the application projected here is a classical one, we showed a situation of high accuracy improvement by just fiddling with the hyperparameters.

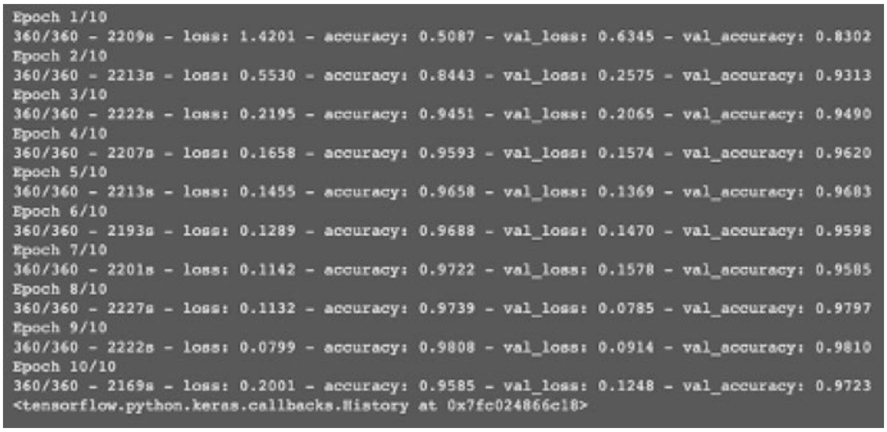
9 Linear Regression Using Keras

Keras library is found to be a high-level API that is used to construct deep learning models due to its simplicity and ease of deployment. Through Keras, a complex deep learning network could be developed with less code.

Problem Statement
Adult overweight and obesity is a common issue that occurs in day-to-day life, and especially during the lockdown period of the pandemic situation, it has occurred worldwide. Obesity is found to be a predominant issue, and the contribution of measurements of BMI alone is not sufficient to support the assessment done by clinicians and achieve obesity-associated health risk in patients [15]. The phenotype of obesity might change over time, which creates an impact in the intensification of abdominal adiposity, as stated in ref. [16]. In ref. [17], as part of categorical analyses, the circumference of the waist is related to health outcomes [18]. Hence, the waist circumference measurement is considered to be a significant metric to augment the health of patients. To analyze it, linear regression could be deployed using Keras [19].

Step 1: Importing the required libraries, necessary modules, and libraries specific to Keras.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
# Import necessary modules
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt
# Keras specific
import keras
from keras.models import Sequential
from keras.layers import Dense
```



```
Epoch 1/10
360/360 - 2209s - loss: 1.4201 - accuracy: 0.5087 - val_loss: 0.6345 - val_accuracy: 0.8302
Epoch 2/10
360/360 - 2213s - loss: 0.5530 - accuracy: 0.8443 - val_loss: 0.2575 - val_accuracy: 0.9313
Epoch 3/10
360/360 - 2222s - loss: 0.2195 - accuracy: 0.9451 - val_loss: 0.2065 - val_accuracy: 0.9490
Epoch 4/10
360/360 - 2207s - loss: 0.1658 - accuracy: 0.9593 - val_loss: 0.1574 - val_accuracy: 0.9620
Epoch 5/10
360/360 - 2213s - loss: 0.1455 - accuracy: 0.9658 - val_loss: 0.1369 - val_accuracy: 0.9683
Epoch 6/10
360/360 - 2193s - loss: 0.1289 - accuracy: 0.9688 - val_loss: 0.1470 - val_accuracy: 0.9598
Epoch 7/10
360/360 - 2201s - loss: 0.1142 - accuracy: 0.9722 - val_loss: 0.1578 - val_accuracy: 0.9585
Epoch 8/10
360/360 - 2227s - loss: 0.1132 - accuracy: 0.9739 - val_loss: 0.0785 - val_accuracy: 0.9797
Epoch 9/10
360/360 - 2222s - loss: 0.0799 - accuracy: 0.9808 - val_loss: 0.0914 - val_accuracy: 0.9810
Epoch 10/10
360/360 - 2169s - loss: 0.2001 - accuracy: 0.9585 - val_loss: 0.1248 - val_accuracy: 0.9723
<tensorflow.python.keras.callbacks.History at 0x7fc024866c18>
```

Fig. 5 Output

Step 2: Source the data.

```
dataset =pd.read_csv('WC_AT.csv')
```

Step 3: The target variable is identified, and the data values are normalized through scaling between 0 and 1.

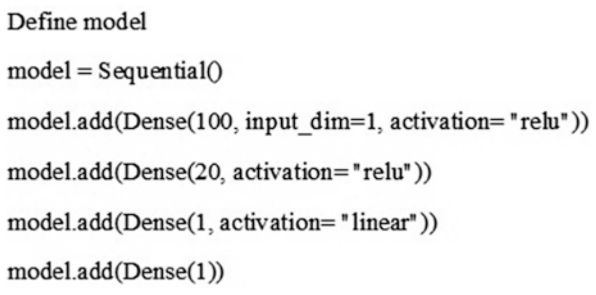
```
target_column = ['AT']

predictors = list(set(list(dataset.columns))-set(target_column))

dataset[predictors] = dataset[predictors]/dataset[predictors]. max()

dataset.describe()
```

Step 4: Fixing the dependent and independent variables in an array for splitting the dataset into a test set and training set in the ratio of 80:20.

```
X = dataset[predictors].values

y = dataset[target_column].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=40)

print(X_train.shape); print(X_test.shape)
```

Step 5:

The sequential model is defined with the linear heap of layers. The first layer is specified with the input dimension as 1, and the ReLu is the activation function. The same process is repeated for the hidden layers. Finally, the output layer is denoted with one node that is determined as the output.



```
Define model

model = Sequential()

model.add(Dense(100, input_dim=1, activation= "relu"))

model.add(Dense(20, activation= "relu"))

model.add(Dense(1, activation= "linear"))

model.add(Dense(1))
```

Step 6:

The optimizer has to be defined along with the loss measure for the training. Here the optimizer used is "adam," and the loss measure is "mean squared error."

There is no need of specifying the learning rate when an adam optimizer is used. The built model has to be fit on the training dataset. The number of epochs mentioned here is 1000 [20].

```
model.compile(loss= "mean_squared_error" , optimizer="adam", metrics=["mean_squared_error"])

model.fit(X_train, y_train, epochs=1000)
```

Output:

```
 Epoch 1/1000 87/87 [==============================] - 1s 9ms/step -
loss: 13968.1219 - mean_squared_error: 13968.1221 Epoch 2/1000 87/87
 [==============================] - 0s 149us/step - loss: 13949.3397
         - mean_squared_error: 13949.3389 Epoch 3/1000 87/87
 [==============================] - 0s 115us/step - loss: 13932.1687
         - mean_squared_error: 13932.1699 Epoch 4/1000 87/87
 [==============================] - 0s 103us/step - loss: 13913.4231
         - mean_squared_error: 13913.4229 Epoch 5/1000 87/87
 [==============================] - 0s 103us/step - loss: 13893.9182
         - mean_squared_error: 13893.9180 Epoch 6/1000 87/87
[==============================] - 0s 92us/step - loss: 13873.9081 -
          mean_squared_error: 13873.9082 Epoch 7/1000 87/87
[==============================] - 0s 92us/step - loss: 13853.0386 -
          mean_squared_error: 13853.0391 Epoch 8/1000 87/87
[==============================] - 0s 99us/step - loss: 13831.0609 -
                     mean_squared_error: 13831.0615

...................................
```

From the output, it has been inferred that the loss function decreases.

Step 7: Model prediction is to be done on the training dataset and test dataset. The root mean squared values are determined.

```
pred_train= model.predict(X_train)

print(np.sqrt(mean_squared_error(y_train,pred_train)))

pred= model.predict(X_test)

print(np.sqrt(mean_squared_error(y_test,pred)))
```

Output:

```
34.38092793923469
25.94993546799555
```

From the results, it has been inferred that the root mean squared error values for the training data and test data are 34.38 and 25.95, respectively. If the root mean squared error value is low, then the model's performance is better [21].

Conclusion
The model shows a stable performance since there is not much variance in the test and training set root mean squared error values. The model could be tested by varying the epochs, neurons in the layers, and adding more hidden layers.

10 Conclusion

This chapter focusses on high-level Python library that are used for deploying deep learning that are deployed on Keras. It provides deep insights into the models of Keras along with the comparison of various frameworks. The process of handling structured and unstructured data has also been explained. Various case studies in binary and multiclass classification have been discussed. Moreover, several models have been deployed with the help of Keras to solve various problems in day-to-day activities.

References

1. Sun, Z.-L., Choi, T.M., Au, K.-F., Yu, Y.: Sales forecasting using extreme learning machine with applications in fashion retailing. Decision Support Systems. **46**(1), 411–419 (2008)
2. Thomassey, S.: Sales forecasts in clothing industry: The key success factor of the supply chain management. International Journal of Production Economics. **128**(2), 470–483 (2010)
3. Teucke, M., et al.: Forecasting of seasonal apparel products. In: Kotzab, H., Pannek, J., Thoben, K.-D. (eds.) Dynamics in Logistics. Fourth International Conference, LDIC 2014 Bremen, Germany, February 2014 Proceedings. Springer, Cham (2014)
4. Yu, X., Yang, L., Huang, X., Aijun: An: mining online reviews for predicting sales performance: a case study in the movie domain. IEEE Trans. Knowl. Data Eng. **24**(4), 720–734 (2012)
5. Nofsinger, J.R.: Social mood and financial economics. J. Behav. Finance. **6**(3), 144–160 (2005)
6. Chan, W.S.: Stock price reaction to news and no-news: drift and reversal after headlines. J. Finance Econ. **70**(2), 223–260 (2003)
7. Strauß, N., Vliegenthart, R., Verhoeven, P.: Lagging behind? Emotions in newspaper articles and stock market prices in the Netherlands. Public Relat. Rev. **42**(4), 548–555 (2016). https://doi.org/10.1016/j.pubrev.2016.03.010
8. Ranco, G., Aleksovski, D., Caldarelli, G., Grčar, M., Mozetič, I.: The effects of twitter sentiment on stock price returns. PLoS One. **10**(9), e0138441 (2015)
9. Bollen, J., Mao, H., Zeng, X.: Twitter mood predicts the stock market. J. Comput. Sci. **2**(1), 1–8 (2011)
10. Luss, R., d'Aspremont, A.: Predicting abnormal returns from news using text classification. Quant. Finance. **15**(6), 999–1012 (2015)
11. Wan, Li, et al.: Regularization of neural networks using dropconnect. International conference on machine learning (2013)
12. Sato, I., Nishimura, H., Yokoi, K.: Apac: Augmented pattern classification with neural networks. arXiv. (2015) preprint arXiv:1505.03229

13. Liang, M., Hu, X.: Recurrent convolutional neural network for object recognition. Proceedings of the IEEE conference on computer vision and pattern recognition(2015)
14. Liao, Z., Carneiro, G.: On the importance of normalisation layers in deep learning with piecewise linear activation units. 2016 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE (2016)
15. Afshin, A., Forouzanfar, M.H., Reitsma, M.B., Sur, P.: Health effects of overweight and obesity in 195 countries over 25 years. N. Engl. J. Med. **377**, 13–27 (2017)
16. Walls, H.L., et al.: Trends in BMI of urban Australian adults, 1980–2000. Public Health Nutr. **13**, 631–638 (2010)
17. Ross, R., Neeland, I.J., Yamashita, S., Shai, I., Seidell, J., Magni, P., Santos, R.D., Arsenault, B., Cuevas, A., Hu, F.B., Griffin, B.A., Zambon, A., Barter, P., Fruchart, J.C., Eckel, R.H., Matsuzawa, Y., Després, J.P.: Waist circumference as a vital sign in clinical practice: a Consensus Statement from the IAS and ICCR Working Group on Visceral Obesity. Nat. Rev. Endocrinol. **16**(3), 177–189 (2020). https://doi.org/10.1038/s41574-019-0310-7
18. JHA, A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing https://doi.org/10.1007/978-3-030-57077-4_2
19. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
20. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Regression, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_4
21. Imambi, S., Prakash, K.B., Kanagachidambaresan, G.R.: pyTorch, programming with tensor flow, EIA/Springer innovations in communication and computing, https://doi.org/10.1007/978-3-030-57077-4_10

Advanced Deep Learning Techniques

Yellapragada Sai Srinivasa Bharadwaj

1 ConvNets

1.1 Introduction to ConvNets

ConvNets is the short form of convolution neural networks that is remarkably similar to ordinary neural networks that you have studied in previous chapters. ConvNets are also made up of neurons with weights and biases as learning parameters. The additional layer(s) that we add to ConvNets is a convolution layer. Convolution is derived from a Latin terminology 'to convolve' which resembles the meaning for rolling together. Mathematically, it is the integral calculation of one function overlapping with another.

$$(f * g)(t) = \int_0^t \sin(t - \tau)\cos\tau \, d\tau$$

The network however still looks like it is taking an image from input layer and provides class scores from the output layer. ConvNet architectures assume that input is an image then proceeds forward to the network; in other words CNN (convolution neural networks) identifies the images by processing on the raw input image, passing through the layers to predict the class from the class scores.

In neural networks, CNN is extensively used for image identification, classification of images, object detection and face detection. One of the real-time uses can be seen in OCR (optical character recognition); for starters OCR is the technique of identifying text from an image. One of the most important OCR is handwritten digit

Y. S. S. Bharadwaj (✉)
K L University, Vaddeswaram, Andhra Pradesh, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_6

recognition [1]. The banking industry make most use out of it for document validation and checking, check approval, credit card processing and many more. Anyone who starts with deep learning, computer vision or machine learning will go through this project at least once; in fact many AI practitioners think of this project as a hello world for deep learning [1].

1.2 Layers

Image is a formation of pixels. A grey scale image is two-dimensional with two channels in the image: a white and a black; for a colour image, there are three channels – red, green and blue – making it a three-dimensional image. In both cases, pixel value is ranged from 0 to 255.

ConvNets architecture can be divided into two stages: feature learning and classification. In general feature learning stage consists of three functions, a convolution function, activation function and a pooling function specifically in this order. They can be repeated multiple times as shown in Fig. 1 [2]. The input layer here is convolution layer itself activated by an activation function. The main objective of this layer is feature extraction; it takes input in [width x height x depth] format where depth depends on type of the image. A grey scale image's depth is 2, and for a colour image the depth is 3. But only a part of that image is considered for convolution, as network may become highly computational if all pixels get into the layer. Activation function determines which neuron can proceed to further layers and which cannot. One of the widely used activation functions is ReLU.

ReLU stands for rectified linear unit. It is computationally efficient as it activates neuron by neuron by converting negative inputs to zero; basically it does not consider negative inputs. When convolution is applied on an image with 128 neurons, 128 feature maps will be generated; each feature map is taken by an activation function to select highlighting features of the input image, but in rare cases, features at the negative region may get saturated and may not proceed further. In such cases we use leaky ReLU activation function. Once features are extracted, the image size is

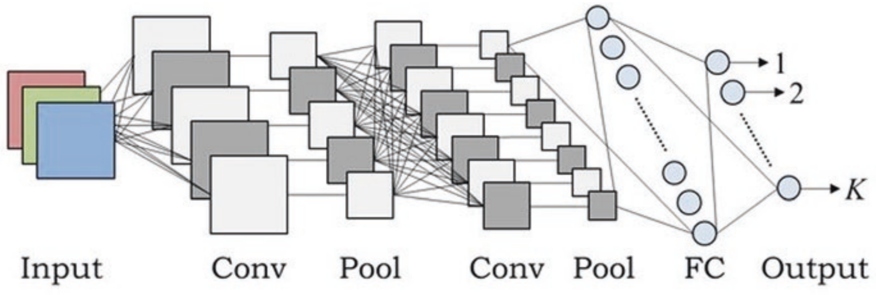


Fig. 1 Basic architecture of ConvNet. (Hidaka et al. [2])

reduced by 1 pixel on each side which means a 28 × 28 input image gets reduced to 26 × 26. This is because we omit bordered pixels while convoluting. Wherever we add a convolution layer, we have to mention four parameters:

1. Number of convolutions
2. Kernel map size
3. Activation function
4. Input shape (only if we consider convolution as input layer)

In Python with TensorFlow, an input convolution layer can be stated as (depending on the platform, the statement may change)

```
keras.layers.Conv2D(64,(3,3),activation='relu',input_shape=
(28,28,1)),
```

A convolution layer with 64 convolution maps of 3 × 3 kernel is activated by ReLU activation function for 28 × 28 grey scale image input. Kernel will get mapped to every 3 × 3 submatrix of the input image such that the output is an enhanced version of the input image as shown in Fig. 2. For each pixel in the input image, a new value is calculated with the help of neighbouring pixels.

$$\begin{aligned}\text{valu}e_{2,2} &= 10*0.1+11*0.2+12*0.3+16*0.4+17*0.5\\ &+18*0.6+22*0.7+23*0.8+24*0.9=58.49\end{aligned}$$

Generalized formula can be given as

$$y[a,b]=\sum_{j=-\infty}^{\infty}\sum_{i=-\infty}^{\infty}x[i,j].h[a-i,b-j]$$

Here, h is the kernel map, and y is the convoluted output of image x.

Features from the image are extracted as feature maps, but these are sensitive towards the location of the input image. Pooling layer will down sample the feature

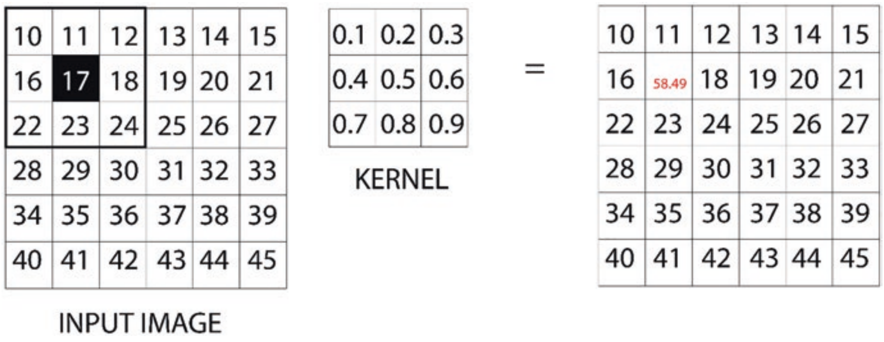


Fig. 2 Convolution operation using kernel

map which in turn reduces the sensitivity. Popular pooling methods are average pooling and max pooling which epitomize the average pixel existence and the maximum pixel existence. Adding a pooling layer to the network after convolution layer is considered to be a common practice. Pooling is an operation like applying a filter on feature map using a pooling operation, and its size is much smaller than the original image or feature map. A 2 × 2 pooling operator reduces feature maps in half as shown in Fig. 3 [3]. Every 2 × 2 submatrix is concise into their maximum equivalent. Similarly, for average pooling, the average of submatrix is evaluated. This method helps in keeping the features while reducing the size of the image. All 64 features with 26 × 26 size from convolution layer can be further reduced such that the resultant feature map can be 13 × 13. We can repeat convolution + pooling layers multiple times for better results.

In classification stage, the two-dimensional feature maps are flattened out into a one-dimensional array or vector to be fed into a classifier. Both training and testing data will go through feature extraction stage. In this way, ConvNets transform the original pixel values, layer by layer, into class scores. A simple ConvNet for handwritten digit recognition is summarized below.

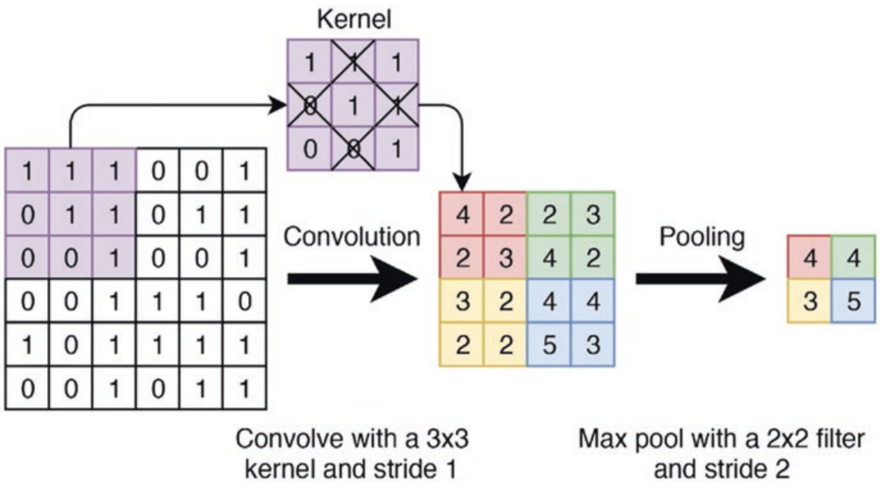


Fig. 3 Demonstration of convolution and pooling. (Verschoof-van der Vaart et al. [3])

```
========================= modelSummary =======================
Model: "sequential"
______________________________________________________________
Layer (type)                 Output Shape              Param #
==============================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
______________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0
______________________________________________________________
flatten (Flatten)            (None, 10816)             0
______________________________________________________________
dense (Dense)                (None, 28)                302876
______________________________________________________________
dense_1 (Dense)              (None, 10)                290
==============================================================
```

In handwritten digit recognition system, the data consists of pixelated images of digits written in different handwriting styles [1]. For training, a subset of the whole data goes through feature extraction phase and classification phase. When a test sample enters the network, the model analyse its features and classifies it based on those features just like any classifier do. In many cases the output layer is activated by a softmax activation function in case of multi-classes and sigmoid activation in case of binary classification. Adding multiple convolution layers will increase the chance of improving prediction rate. The deeper the network is, the complex it becomes to deal with the features. Popular versions of ConvNets are ResNet, AlexNet, VGG16 and Inception Networks which are available in public platform.

Building a ConvNet in Python using Keras is straightforward and does not need any calculations. The first step is to obtain the data and preprocess it to the needs of our network.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

handwriting_mnist = keras.datasets.mnist
(train_images,train_labels),(test_images,test_labels) = handwrit-
ing_mnist.load_data()
train_images = train_images.reshape(len(train_images),28,28,1)
test_images = test_images.reshape(len(test_images),28,28,1)
```

The structure of ConvNet used here is a pair of convolution and max pooling layers as in the model representation above.

```
model = keras.models.Sequential([                          keras.
layers.Conv2D(64,(3,3),activation='relu',input_shape=(28,28,1)),
keras.layers.MaxPool2D(2,2),
keras.layers.Flatten(),                                 keras.lay-
ers.Dense(28,activation='relu'),
keras.layers.Dense(10,activation='softmax')
])

m o d e l . c o m p i l e ( o p t i m i z e r = ' a d a m ' , m e t r i c s =
['acc'],loss='sparse_categorical_crossentropy')


class handwriting_acc_callback(keras.callbacks.Callback):
  def on_epoch_end(self,epoch,logs={}):
    if(logs.get('acc')>=0.9):
   print("\n Model has reached 90% accuracy! Congratulations !!!!!")
       self.model.stop_training = True

handwriting_acc_callback = handwriting_acc_callback()


model.fit(train_images,train_labels,epochs=100,callbacks=[handwrit
ing_acc_callback])
model.evaluate(test_images,test_labels)
```

Callback here will restrict the train from overfitting; we will see more about this as a standalone topic.

1.3 Construction and Architecture

Convolution layer is the main building block of ConvNets that almost does all the extensive work. Every filter is spatially small but extends throughout the image. When dealing with high- dimensional resources such as images, we connect each neuron to a local region of input space. During the forward propagation, we cross each filter throughout the image to compute the dot product just like we have done previously, but when considering the neural connectivity there a hyper parameter dealing with special extent, i.e. receptive field of neuron which is nothing but the size of the filter. The approach of that filter is not just limited to width or height, it connects to the depth also; in fact the connectivity extends to the entire depth of the image.

For example, if an RGB CIFAR-10 image of size 32 × 32 × 3 is convoluted with 5 × 5 filter, then each neuron in the layer has the same weight as 5 × 5 × 3 in the input region. Therefore, a total of 75 connections are made with the input volume.

Connections from convolution layer to input layer is simply straightforward compared with the arrangement of neurons for output. There are three parameters responsible for controlling the dimensions of output:

1. **Depth**: it resembles the number of filters used, and for each cycle, it feeds a different input. For example, if the first cycle of convolution inputs a raw image, then other neurons in the same space may be activated by any property of that image like edges or colours.
2. **Stride**: this decides the jump ratio of filter across pixels. For stride = 1, the filter gets moved by one pixel at a time. The higher the stride, the lower the output special volume.
3. **Zero-padding**: like we saw previously, convolution omits the bordering pixels. Zero-padding allows us to control the borders by filling them with zero; thus, the image size will not be reduced.

Using the function of input volume size (W), the convolution field size (F), the stride (S) and the zero-padding level (P), output space neuron fit can be given as $(W - F + 2P)/S + 1$. For example, a 7×7 input image and a 3×3 kernel/filter with 1 and 0 as stride and padding, respectively, can produce a 5×5 output [4].

In a larger network, controlling these parameters is a troublesome and confusing task. So we use a parameter sharing scheme to control the number of parameters influencing the network in a whole. For example, the architecture that won the ImageNet Challenge in 2012 trained the network with $227 \times 227 \times 3$ images. The first convolution layer used the neurons with $F = 11$, $S = 4$, $P = 0$, $K = 96$; therefore, output size can be calculated as $(227 - 11)/4 + 1 = 55$, 1 is added because of bias. The number of neurons in the first convolution layer can be $55 * 55 * 96 = 290, 400$. The number of weights in each layer can be calculated as $11 * 11 * 3 = 363$. Together they give $290400 * 364 = 105, 705, 600$ parameters for the first layer alone. To reduce the parameters, we can denote a two-dimensional depth slice, i.e. by considering 96 unique sets of value compositions like weights and biases, i.e. $96 * 11 * 11 * 3 = 34, 944$ unique parameter weights and 96 unique biases [5].

Programmatically, for an input shape $11 \times 11 \times 4$ with $P = 0$, $F = 5$, $S = 3$, the output volume is $(11 - 5)/3 + 1 = 3$, so the activation map V (output of particular convolution layer) can be formulated as

```
V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0
V[1,0,0] = np.sum(X[3:8,:5,:] * W0) + b0
V[2,0,0] = np.sum(X[5:10,:5,:] * W0) + b0
V[3,0,0] = np.sum(X[7:12,:5,:] * W0) + b0
```

Here the w0 is the weight vector of shape $5 \times 5 \times 4$, and we are using the same weights and bias (parameter sharing). Multiple activation maps can be formed by going along other dimensions like

```
V[0,0,1] = np.sum(X[:5,:5,:] * W1) + b1
V[1,0,1] = np.sum(X[3:8,:5,:] * W1) + b1
V[2,0,1] = np.sum(X[5:10,:5,:] * W1) + b1
V[3,0,1] = np.sum(X[7:12,:5,:] * W1) + b1
```

Play with this to see how they reflect the output. Try changing the limiting pixels of the image *X* by going along different axes and dimensions. For example, `V[0,1,1] = np.sum(X[:5,3:8,:] * W1) + b1` (example of going along y), `V[1-3] = np.sum(X[5:10,7:12,:] * W1) + b1` (going along both).

In short, the convolution layer accepts $W_1 X\ H_1 X\ D_1$ volume size with four parameters (K, F, S, P) producing a $W_2\ X\ H_2\ X\ D_2$ where $W_2 = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)/S + 1$ and $D_2 = K$. The common hyperparameter values are $F = 3$, $S = 1$, $P = 1$.

As it is a common practice to include a pooling layer in between successive convolution layers, it also contains some characteristics. This layer operates on its own by resizing every depth slice in the input. The popular filter size for pooling is 2 × 2 with stride size 2. Despite its down sampling methodology of discarding almost 75% of the activation, the depth remains the same. Output from the convolution layer is treated as input here with $W_1 X\ H_1 X\ D_1$volume size, accepting two parameters F and S producing $W_2 X H_2 X D_2$ where $W_2 = (W_1 - F)/S + 1$, $H_2 = (H_1 - F)/S + 1$ and $D_2 = D_1$.

2 RNN, LSTM and GRU

The popularity of neural networks is being influenced by advancements in mobile technologies like Apple's Siri, Amazon's Alexa and Microsoft's Cortana. These voice assistants can remember the input to process an upcoming query. For example, a query 'What food is famous in Italy?' was asked by a user, and the assistant replied with an appropriate answer as the query is straightforward. If the user asks another query like 'How is the weather there?', this query is unclear as the keyword there is not relevant; here where RNN comes into picture. RNN stands for recurrent neural networks; this algorithm remembers its input because of an internal memory logic making it suitable for understanding and predicting sequential data structures. From the previous example, the assistant can now understand that the term 'there' is not relevant, but it is relative to the previous query where the subject refers to the term 'Italy'. Therefore, the network updates the query to 'How is the weather Italy?'. Because of this capability in RNN, it can also predict the next sequence of queries very precisely. This is why many AI practitioners prefer RNN for data like financial data, audio, video, time series, weather and much more over other algorithms.

In a feed-forward network like ConvNets, the information is unidirectional (from input, through hidden, to output) without visiting a neuron/node twice. Unlike them, RNN loops the information in cycles, and while taking a decision it considers the

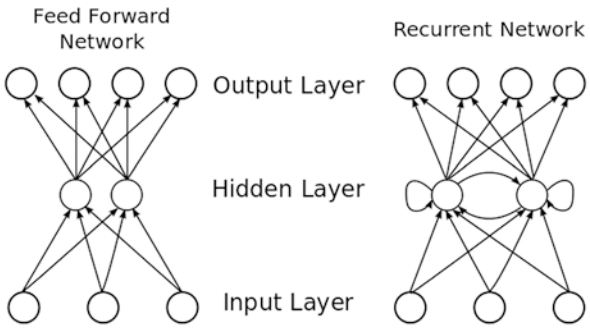


Fig. 4 Feed-forward network vs RNN. (Dana and Correll [6])

present cycle which already experienced the previous knowledge or information as shown in Fig. 4 [6]. The one important note while understanding RNN is that they apply weights to current as well as previous input; also, the adjustment of weights is carried out through gradient descent and backpropagation. Unlike feed-forward network, RNN is capable of mapping transactions from one to many, many to one and many to many neurons. To get a clear understanding about RNN, you'll need to understand how a normal feed-forward network and sequential data work. You have already seen the working of feed-forward network previously, so now let's look into sequential data.

Sequential data is the data arranged in an ordered manner but not necessarily in a chronological order. Examples of sequential data are customer purchase data in a store, web history, career growth and many more. A time series data is also a sequential data having successive intervals of time. A best example for time series data is weather data. A typical RNN contains a short memory which is capable of remembering the immediate past, but for longer sequences it is necessary to access data from unknown duration; for such applications we use LSTM along with RNN. LSTM stands for long short-term memory, and they are designed especially to address long-term dependency problem in traditional RNN. All recurrent networks form a connection or chain of repeating modules; traditionally this repeating module is a single tanh layer, but in LSTM the repeating module consists of four interactive layers as show in Fig. 5 [7]. Mathematically, the gates in LSTM cell can be described as

$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$

$$f_t = \sigma\left(x_t U^f + h_{t-1} - 1W^f\right)$$

$$o_t = \sigma\left(x_t U^f + h_{t-1} - 1W^f\right)$$

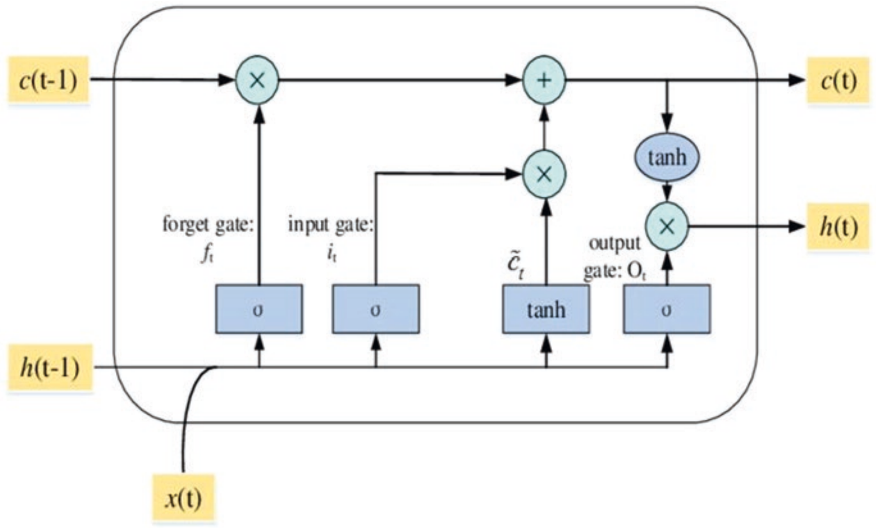


Fig. 5 Structure of LSTM. (Yuan et al. [7])

$$\widetilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$

$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \widetilde{C}_t\right)$$

$$h_t = \tanh\left(C_t\right) * o_t$$

where W is the recurrence between the current and previous nodes. U contains weights from input for hidden layers, $\tilde{C}$ is called candidate's hidden state and C is the memory unit.

Backpropagation in RNN is called 'Backpropagation Through Time'. While constructing a RNN using any framework, backpropagation is taken care automatically, but understanding it is important to debug any issue while running the model. The weight transfer in RNN is usually referred in rolls; a forward roll is called a row, whereas the backward roll is called unroll. The change in the outputs of current roll to the next roll through a function is described as gradient. Mathematically a gradient is the partial derivative of the function with respect to its inputs. The higher the gradient, the better the performance of the model learning. Two key issues in standard RNN are exploding gradient problem and vanishing gradient problem. Exploding occurs when the algorithm assigns higher weights to insignificant details, which can be solved by squashing or omitting some gradients. When the gradients are too small to consider, then it is called vanishing gradient problem; this problem can be solved using LSTM as it keeps the steep to a threshold level to maintain the training minimal and accurate. The process of LSTM is interactive because of the inclusion of cell state. Cell state is like a flow of work where the activation

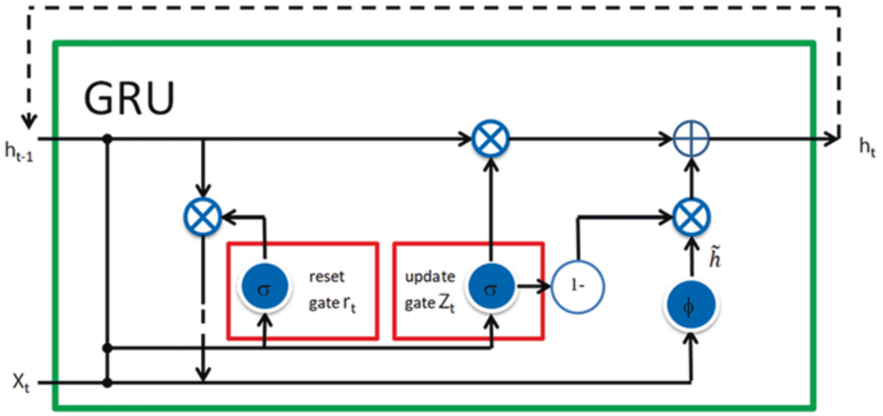


Fig. 6 Structure of GRU. (Su and Kuo [8])

function's output joins the main stream of the respective network knowing what state is current state and what state is previous state. Building a LSTM is so easy that it only involves a few tensor operations and one loop.

A smaller version of LSTM with almost the same functionality is GRU (gated recurrent unit). The main reason behind the development of GRU is to solve vanishing gradient problem. GRU is an improved version of standard RNN; it contains only two gates, an update gate and a reset gate. These two gates will decide which information will get through the network and which information is irrelevant by using only four operations as shown in Fig. 6 [8]. The update gate (Z_t) is responsible for passing past information. Update gate z_t for time step t can be given as

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

where x_t is the input where weights $W(z)$ get multiplied to it when plugged to the network and h_t is the candidate activation with respect to time. Like x_t, the previous candidate's activation when plugged into the network, it gets multiplied by its own weights $U(z)$. Both results add up together and are given to a sigmoid activation function to sustain the resultant in between 0 and 1. This operation eliminates the risk of occurring vanishing gradient problem. The other gate is reset gate which is responsible for clearing the memory (decide how much past information to forget); the degree of forgetful can be calculated using

$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

GRU's output is influenced by both current memory content and current time step. The current memory output h_t' can be achieved by restoring the information from the past using

$$h_t' = \tanh\left(Wx_t + r_t \odot Uh_{t-1}\right)$$

The importance of current memory is linked with the effect of reset gate. Hadamard product ($\odot$) between reset gate and product of previous memory content and its weight Uh_{t-1} will determine what information should be removed. Finally, the sum of Hadamard product of update gate with previous memory content and Hadamard product of update gate's zeroth portion ($1 - Z_t$) and current memory content

$$h_t = z_t \odot h_{t-1} + \left(1 - z_t\right) \odot h_t'$$

Since the model is not taking out the inputs every time a new information passes through out network. ConvNets are considered to be superior to recurrent networks. We have already seen ConvNets for image classification task; now let's see how convolution helps in processing sequential data.

3 Sequence Processing Using ConvNets

Previously we have learned about convolution and how it helps in solving computer vision problems. The characteristics of convolution like feature extraction from input space of 2D image made ConvNets flourish in computer vision. These same properties can further be extended to sequence processing as time itself can be treated as a special dimension. Usually the one-dimensional convolutions are computationally higher than recurrent networks, but recent research in the field of audio translation and generation used the dilated kernel which is prone to have a better result compared to RNN for simple tasks like forecasting and text classification. What we have seen previously is 2D convolution that extracts 2D feature maps by applying a 2D filter or kernel. In the same way we can extract 1D sequence using 1D convolutions. The main idea is the same as 2D convolution; feature maps are obtained from a time patch in the input. The transformation applied on the patch (a part of the input) is the same throughout the sequence – thus, a pattern from one position can be recognized at another making; this is called translation invariance. For example, let's say a 1D ConvNet is processing a sequence of symbols or characters using a window size of 5; that means the network is able to learn a patch of the sequence of length 5 aka fragment. Whenever it came across this fragment again, the network will be able to recognize these words in the input sequence to build a context. This understanding of fragments and segments in the sequence for building a context is called word morphology.

Like 2D ConvNets, 1D ConvNets also have a pooling layer associated with convolution layer, and the operation of 1D pooling is equivalent to 2D pooling, outputting the maximum or average value from the patch (subsequence). In terms of sequences, pooling can also be referred as subsampling. Building a 1D ConvNet in Python using Keras is identical to 2D ConvNet. First we need to obtain the data and preprocess it

```
from keras.datasets import imdb
from keras.preprocessing import sequence
max_features = 100
max_length = 5

print('Loading data...')
(x_train, y_train),(x_test, y_test) = imdb.load_data(num_words =
max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=max_length)
x_test = sequence.pad_sequences(x_test, maxlen=max_length)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

In 2D ConvNet we stacked a pair of convolution layers with max pooling 2D layer. Here that changes to max pooling 1D layer, and one major change is the kernel size. In 2D network we mention the kernel size as 3 × 3 or 5 × 5 which results in 9 or 25 pixels, respectively, but here in 1D a 3-window size will only contain 3 features which is considerably smaller so we can increase the size to our wish. 7 as window size is optimal for this program.

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features,                     128,
input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer=RMSprop(lr=1e-4),   loss='binary_crossen-
tropy',             metrics=['acc'])
history = model.fit(x_train, y_train, epochs=10, batch_size=128,
validation_split=0.2)
```

Comparing to LSTM, 1D ConvNet's runtime is higher on both CPU and GPU. In the same way 2D convolution performs well on images, 1D convolution performs well on temporal data. Because RNN is computationally expensive for processing long sequences, 1D CNN can be used as a preprocessing step to compress the sequence for feeding into RNN.

4 Keras Callbacks and TensorBoard

4.1 Callbacks

Building a model, whether it be a CNN or RNN, is only a quarter of the work done; the rest of it is debugging and troubleshooting the model. One of the greatest debugging tools provided by Keras is callbacks. These help to understand the working of the code while it is running. In other words, we can retrieve the statistical and internal state information of the model during the training itself. Callback is not a layer or a separate model; it is a function that we call on fitting. We can write our very own callback if needed, but in most of the cases it is not necessary as Keras offers a variety of callback functions [19].

BaseLogger and History
This is a basic callback applied to the model by default. When fitting the model, the general statement is `model.fit();` this does not assign results to anything, but when we assign the same statement to a variable, then that variable is assigned to an object `keras.callbacks.history` containing each epoch's average accuracy and loss information as a dictionary.

Model Checkpoint
After successful epoch, this callback will save the model as a checkpoint in HDF5 format with a name of our desire. This callback is helpful when we are running our model in cloud and that is taking more time.

CSVLogger
Like the name suggests, this callback writes the epoch information or logs about epoch, accuracy and loss into a CSV file. This helps in understanding the behaviour of the model while training, validating and testing. This will help when we are comparing different models for an application.

EarlyStopping
Overfitting is one of the major problems in machine learning; it is equivalent to overconfidence in human beings. EarlyStopping callback will terminate the training before it is completed. We can specify the learning metric to a value that helps this callback to understand the change in shift in the metric.

RemoteMonitoring
This callback's main intention is to integrate machine learning models with web services. This callback uses HTTP POST to send status in JSON format that helps in monitoring and handling specific tasks remotely.

LearningRateSchedular
Optimization in neural networks is crucial for minimizing the costs and maximizing the fitness. This callback helps in tweaking the learning rate of optimizers. One of such optimizers is gradient descent; this callback helps in emphasizing the step size needed to be taken during the process.

LambdaCallback
A custom callback, if you did not find a suitable callback for your requirement, then this is your option. This lets you create a basic callback that returns epoch, batch or training information. For a complex callback, you have to inherent keras.callbacks. Callback class. This lets you have the control over execution of callback.

All the above callbacks help in building a model effectively and quickly. There is no restriction of using one callback per model; you can use any number of callbacks you want. There is one more callback which is a favourite of many AI practitioners or those who are familiar with TensorFlow: TensorBoard.

TensorBoard
A beloved callback where logs are written to a directory which further lets you use them in TensorFlow's visualization tool is named TensorBoard. The highlighting factor is that it also works with TensorFlow alternatives like CNTK or Theano.

4.2 TensorBoard

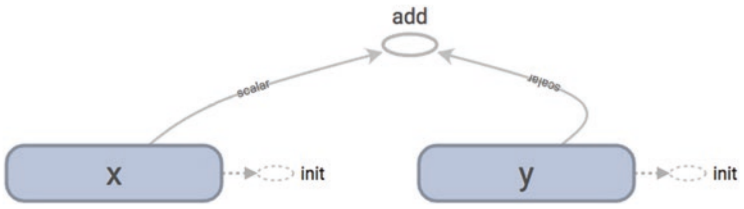
TensorFlow is a great place to start building machine learning models with user-friendly Keras API. It was developed by Google as a framework that utilizes graphical representations for effective computation. A program in TensorFlow contains two phases: one being the addition of node, and the other is feeding nodes with inputs for graph execution. For example, addition of two numbers in TensorFlow looks like this:

```
x = tf.Variable(3, name = 'x')
y = tf.Variable(4, name= 'y')
sum = x + y
print(sum)
```

By running the above code, you may think we get 7 as output, but instead we will end up with this:

```
Tensor("add_1:0", shape=(), dtype=int32)
```

As mentioned earlier, we execute the graph by feeding inputs. TensorFlow will construct a graph from the nodes and the operations we want to compute. For the above code, the graph will look like this:



This visualization can be seen in TensorBoard. TensorBoard provides tools for measurement and visualization during workflow. It is not different from what we have seen earlier. We just have to add a callback to the fitting statement that creates logs specifically for TensorBoard. In Python, we need to import the tensorboard library first:

```
from tensorflow.keras.callbacks import TensorBoard
```

Let's build a model that can classify cats and dogs:

```
NAME = "cats-vs-dogs-cnn-64x2".format(time.time)
tensorboard = TensorBoard(log_dir="logs/{}".format(NAME))
```

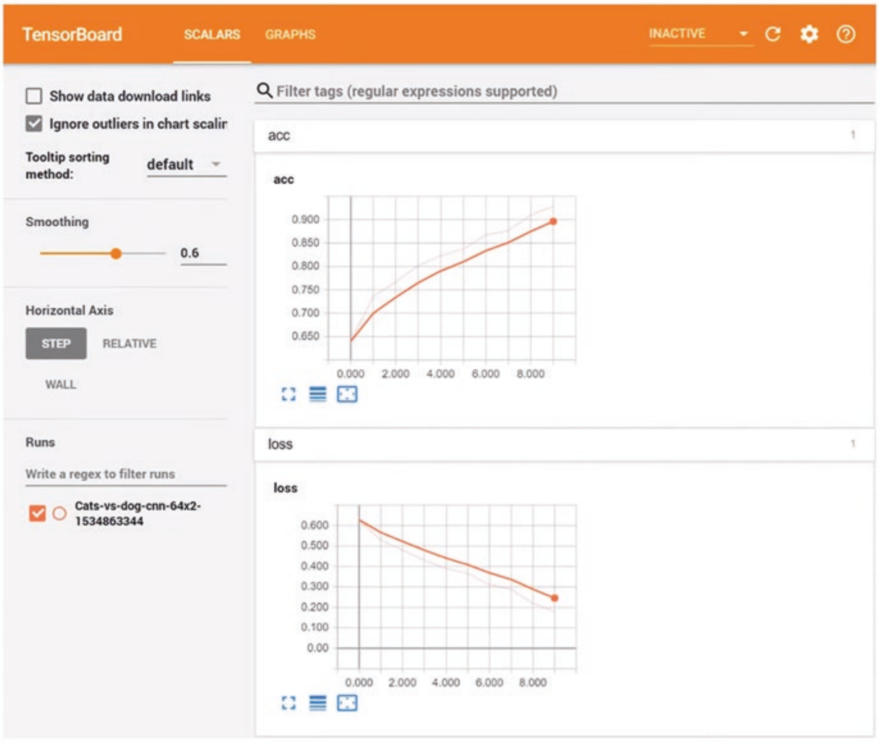
Now, the tensorboard variable is the model which is readable by TensorBoard, but it does not contain any information in it. We can pass our training information to it using the callback that we have seen earlier:

```
model.fit(X, y,
          batch_size=32,
          epochs=3,
          validation_split=0.3,
          callbacks=[tensorboard])
```

The callback provided here is a list. We can provide other callbacks in the list as well. Now let's build our model:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from    tensorflow.keras.layers    import    Dense,    Dropout,
Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
import pickle
import time
pickle_in = open("X.pickle","rb")
X = pickle.load(pickle_in)

pickle_in = open("y.pickle","rb")
y = pickle.load(pickle_in)

X = X/255.0


model = Sequential()


model.add(Conv2D(256, (3, 3), input_shape=X.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(256, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())  # this converts our 3D feature maps to 1D
feature vectors
model.add(Dense(64))

model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'],
              )
```

If you check your program directory, there will be a new folder created named logs; that folder consists of another directory with the name we have provided – in this case it is 'cats-vs-dogs'. To get a view of that log, we need to access command line from the main directory (the location where log folder exists), and then we need to trigger the TensorBoard with the following command:



☐ Show data download links
☑ Ignore outliers in chart scalir
Tooltip sorting method: default
Smoothing 0.6
☑ Cats-vs-dog-cnn-64x2-1534863344

Fig. 7 Cats vs dogs TensorBoard dashboard

```
tensorboard –logdir='logs/'
```

After a successful execution of the above command, we get a URL that will take you to the TensorBoard visualization dashboard that looks like in Fig. 7.

Beginners usually visualize their epochs using matplotlib which is a powerful visualization tool without any doubt, but this TensorBoard provides many functionalities like filtering graphs, regular expressions and running samples without re-running the model. Monitoring the model behaviour during training is a great way of troubleshooting the problem; with tensorboard we can do that too. Let's say you want to monitor the distribution of weight units. We can add a tf.summary.histogram to our desired output. This procedure is called custom logging. For example, if you want to visualize the gradient descent step change which is implemented using LearningRateScheduler callback, we can do that by attaching the tf.summary.scalar to the scheduler function as shown in the code below:

```
logdir       =       "logs/scalars/"       +       datetime.now().
strftime("%Y%m%d-%H%M%S")
file_writer = tf.summary.create_file_writer(logdir + "/metrics")
file_writer.set_as_default()

def lr_schedule(epoch):
  """
 Returns a custom learning rate that decreases as epochs progress.
  """
  learning_rate = 0.2
  if epoch > 10:
    learning_rate = 0.02
  if epoch > 20:
    learning_rate = 0.01
  if epoch > 50:
    learning_rate = 0.005

      tf.summary.scalar('learning   rate',   data=learning_rate,
step=epoch)
  return learning_rate

lr_callback = keras.callbacks.LearningRateScheduler(lr_schedule)
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)

model = keras.models.Sequential([
    keras.layers.Dense(16, input_dim=1),
    keras.layers.Dense(1),
])

model.compile(
    loss='mse', # keras.losses.mean_squared_error
    optimizer=keras.optimizers.SGD(),
)

training_history = model.fit(
    x_train, # input
    y_train, # output
    batch_size=train_size,
    verbose=0, # Suppress chatty output; use Tensorboard instead
    epochs=100,
    validation_data=(x_test, y_test),
    callbacks=[tensorboard_callback, lr_callback],
)
```

If we look into the board again (Fig. 8),

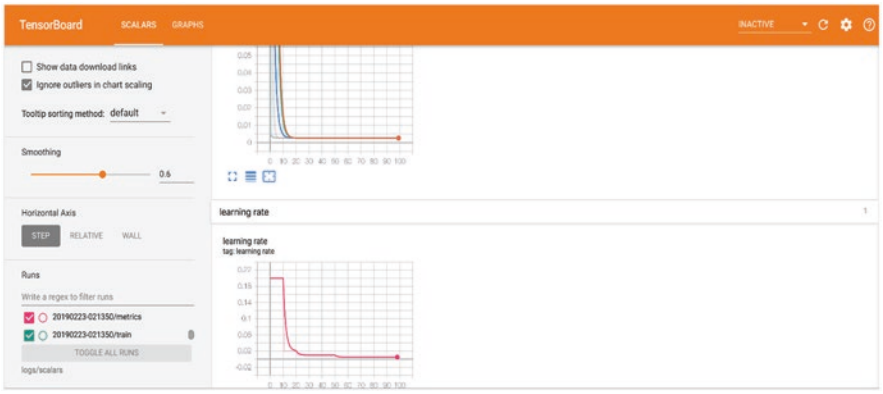


Fig. 8 Learning rate visualization in TensorBoard

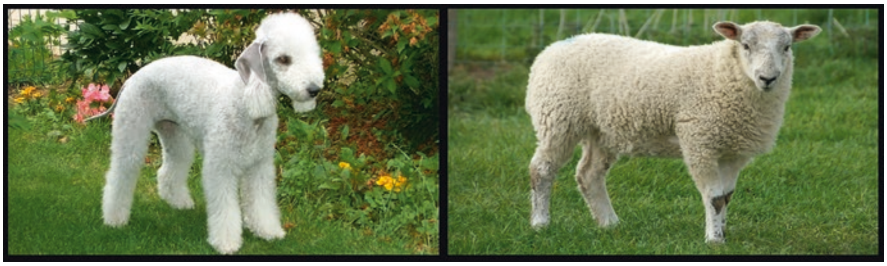


Fig. 9 Non-prototypical dog vs sheep

When performing iterative models, TensorBoard comes handy. Many AI practitioners use TensorBoard for understanding the data as well. The more you understand your data, the better the model you build.

5 Deep Dream and Neural Style Transfer

Previously we have seen the working of CNN (ConvNets) for classifying an image. Image classification is a big problem because it is extremely hard for machines to differentiate between objects of different sorts. One platform where you can experience image classification is Google Image Search. Here anyone can upload an image to identify what it is, but as you can see, it is not perfect. Statistics says that humans take a little time detecting a different dog that we have not seen before, and even further when the dog is non-prototypical like shown in Fig. 9, it gets even harder to classify [9]. When classifying dogs and their breeds, humans can do that with a little hesitation, but machines fail as the geometric features are super similar making it hard to isolate as shown in Fig. 10. To solve this problem, researchers

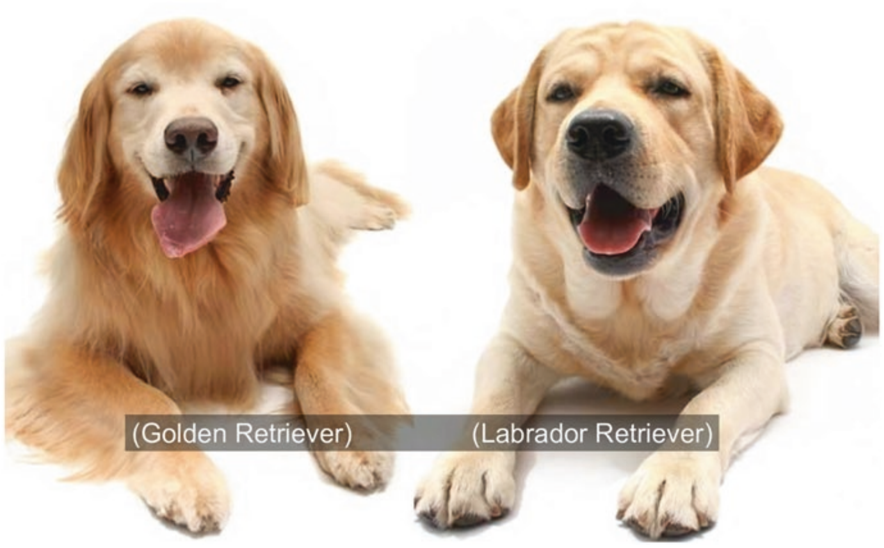


Fig. 10 Golden retriever vs Labrador retriever

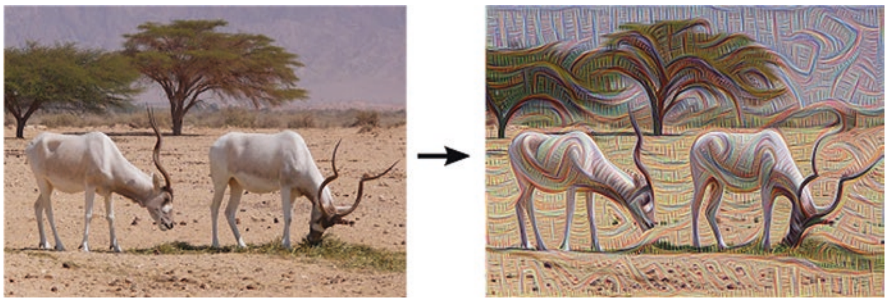


Fig. 11 Output from the first layer

came up with a solution where they started looking into the output of each hidden layers of every image fed into the network and labelling them as training data to feed them back into the network. Basically, they are adding processed image at every layer and label them according to their original image into the training dataset, thus forming millions of images for a single dog breed image. This experiment is called 'deep dream'.

Features are the most important part in image classification. In Fig. 10 the features of both dogs are fairly similar, thus fooling the network into false beliefs. With deep dream, we can use features as image to extract more features from them; this way we can obtain more isolate features from images.

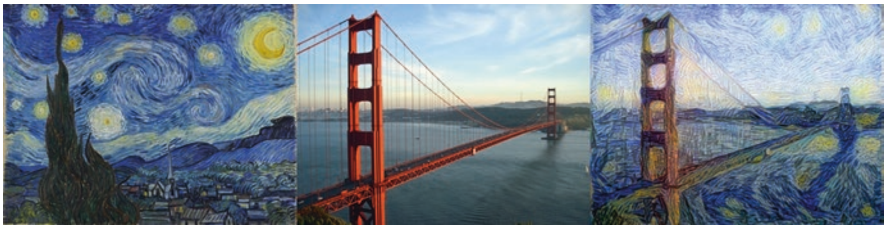


Fig. 12 An example of neural style transfer technique. (Chen et al. [10])

Each layer in a network deals with features of different abstraction, so the complexity of features depends on the choose layer. An image went through the first layer that looks similar to Fig. 11. The first layer usually extracts hard stokes and simple patterns from the image as they are sensitive towards edges. Going further into the network, layers extract complex features to a minute detail. If the network went through a loop, applying its output to itself as input, an extreme stream of network knowledge can be obtained. When passing an image through the pretrained model of massive knowledge in feature extraction and applying its features to the passed image, we can observe a different styling applied to the image which is called neural style transfer technique, NST in short.

NST transfers the patterns of one image and applied it onto another. For example, let's take an image of a bridge as a content image and The Starry Night by Vincent van Gogh as style image. The resultant image is a mixture of both the images as shown in Fig. 12 [10].

Regulating the measure of style is depended on optimizing the network. However, the basic idea is to take two images (style and content image) and figure out their speciality. Specifically for the content image, the details shouldn't be removed without changing the image's geometry features. From the styling image, abstract features are to be extracted like brush patterns, colour strokes, tone, etc. Once it's done, we feed both of these into a network that combines these two images into one. But not all combination of images gives great results; some combinations make a mess out of it. The main use case of NST is changing the weather. For example, it can be able to make spring time images look like winter and summer images look like spring. It can even change day to night and vice versa.

Implementation of NST can be done in five simple steps:

1. Visualizing the data
2. Preprocessing
3. Loss function
4. Model creation
5. Optimizing the loss function

Here we are using a pretrained VGG19 model. VGG19 model consists of a lot of layer which helps in a better style extraction. Before starting the actual process, let's set up the environment first.

Setup

The structure of this technique works better in TensorFlow's imperative programming environment (eager execution) that evaluated operations quicker.

```
tf.enable_eager_execution()
print("Eager execution: {}".format(tf.executing_eagerly()))
```

You can use any image of your wish. Here, I'm giving some sample images and a way to download them directly into the runtime environment in Google's Colab:

```
import os
img_dir = '/tmp/nst'
if not os.path.exists(img_dir):
    os.makedirs(img_dir)
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/d/d7/Green_Sea_Turtle_grazing_seagrass.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/0/0a/The_Great_Wave_off_Kanagawa.jpg


!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/b/b4/Vassily_Kandinsky%2C_1913_-_Composition_7.jpg


!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/0/00/Tuebingen_Neckarfront.jpg


!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/6/68/Pillars_of_creation_2014_HST_WFC3-UVIS_full-res_
denoised.jpg


!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipe-
dia/commons/thumb/e/ea/Van_Gogh_-_Starry_Night_-_Google_Art_
Project.jpg/1024px-Van_Gogh_-_Starry_Night_-_Google_
Art_Project.jpg
```

By now you should know that packages are necessary for Python, so let's go ahead and import some necessary packages:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (10,10)
mpl.rcParams['axes.grid'] = False

import numpy as np
from PIL import Image
import time
import functools

%tensorflow_version 1.x
import tensorflow as tf
from   tensorflow.python.keras.preprocessing   import   image   as
kp_image
from tensorflow.python.keras import models
from tensorflow.python.keras import losses
from tensorflow.python.keras import layers
from tensorflow.python.keras import backend as K
```

Let's complete the setup by specifying the content and style images. Here, I'm using the turtle image as content image and the great wave of kanagawa as style image:

```
content_path = '/tmp/nst/Green_Sea_Turtle_grazing_seagrass.jpg'
style_path = '/tmp/nst/The_Great_Wave_off_Kanagawa.jpg'
```

Now let's get started by visualizing our image data. For that, let's create two functions: load_img and imshow:

```
def load_img(path_to_img):
  max_dim = 512
  img = Image.open(path_to_img)
  long = max(img.size)
  scale = max_dim/long
     img   =   img.resize((round(img.size[0]*scale),   round(img.
size[1]*scale)), Image.ANTIALIAS)
  img = kp_image.img_to_array(img)
  img = np.expand_dims(img, axis=0)
  return img
```

This function will preprocess the image and assign the scaled image to a variable:

```
def imshow(img, title=None):
  out = np.squeeze(img, axis=0)
  out = out.astype('uint8')
  plt.imshow(out)
  if title is not None:
    plt.title(title)
  plt.imshow(out)
```

This function will plot the image passed to it. Now let us plot the images:

```
plt.figure(figsize=(10,10))
content = load_img(content_path).astype('uint8')
style = load_img(style_path).astype('uint8')
imshow(content, 'Content Image')
imshow(style, 'Style Image')
plt.show()
```

Output will be displayed in a graph style as in Fig. 13.

The next step is preprocessing the data. Like mentioned earlier we are using VGG19 model here. This model comes with its own preprocessing method:

```
def load_and_process_img(path_to_img):
  img = load_img(path_to_img)
  img = tf.keras.applications.vgg19.preprocess_input(img)
  return img
```

The content loss function determines the distance from output (x) and content (p) images and $C_{nn}(X)$ be the network VGG19. The VGG19 model contains many layers for classification, but our target is not to classify images but to extract style in the image to apply to another image. So let's pull out necessary layers from the model. Let $F_{ij}^{l}\left(x\right)$ be the intermediate layer representing features of x and $P_{ij}^{l}\left(p\right)$ be the

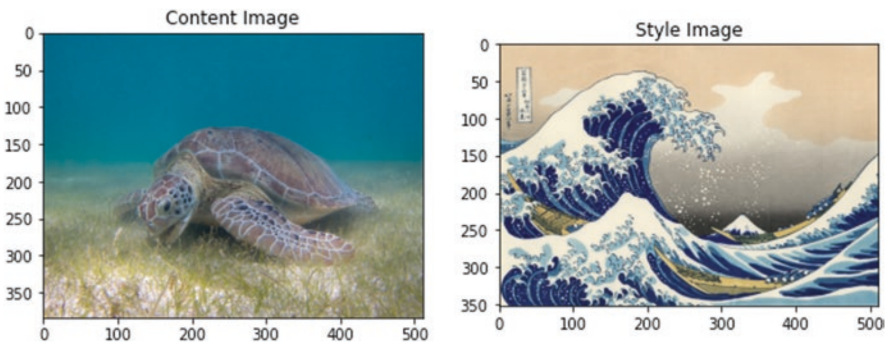


Fig. 13 Visualizing the content and style image for neural style transfer

intermediate layer representing features of *p* at layer *l*. Therefore, the loss function can be described as

$$L_{\text{content}}^{l}\left(p,x\right)=\sum_{i,j}\left(F_{ij}^{l}\left(x\right)-P_{ij}^{l}\left(p\right)\right)^{2}$$

```
def get_content_loss(base_content, target):
  return tf.reduce_mean(tf.square(base_content - target))
```

The total style loss can be given as

$$E_l=\frac{1}{4N_l^2M_l^2}\sum_{i,j}\left(G_{ij}^{l}-A_{ij}^{l}\right)^{2}$$

However, the contribution of each layer's loss can be given as

$$L_{style}\left(a,x\right)=\sum_{l\in L}w_lE_l$$

```
def gram_matrix(input_tensor):
  channels = int(input_tensor.shape[-1])
  a = tf.reshape(input_tensor, [-1, channels])
  n = tf.shape(a)[0]
  gram = tf.matmul(a, a, transpose_a=True)
  return gram / tf.cast(n, tf.float32)

def get_style_loss(base_style, gram_target):
feature map and the number of filters
  height, width, channels = base_style.get_shape().as_list()
  gram_style = gram_matrix(base_style)

   return tf.reduce_mean(tf.square(gram_style - gram_target))# /
(4. * (channels ** 2) * (width * height) ** 2)


content_layers = ['block5_conv2']
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1'
               ]
num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

With loss function being implemented, let us build a model with only intermediate layers from the VGG19 layers:

```
def get_model():
    vgg = tf.keras.applications.vgg19.VGG19(include_top=False,
weights='imagenet')
  vgg.trainable = False
    style_outputs = [vgg.get_layer(name).output for name in
style_layers]
    content_outputs = [vgg.get_layer(name).output for name in
content_layers]
  model_outputs = style_outputs + content_outputs
  return models.Model(vgg.input, model_outputs)
```

The last part is to optimization:

```
import IPython.display


def run_style_transfer(content_path,
                       style_path,
                       num_iterations=1000,
                       content_weight=1e3,
                       style_weight=1e-2):
  # We don't need to (or want to) train any layers of our model, 
so we set their
  # trainable to false.
  model = get_model()
  for layer in model.layers:
    layer.trainable = False

  # Get the style and content feature representations (from our 
specified intermediate layers)
    style_features, content_features = get_feature_
representations(model, content_path, style_path)
  gram_style_features = [gram_matrix(style_feature) for style_fea-
ture in style_features]

  # Set initial image
  init_image = load_and_process_img(content_path)
  init_image = tf.Variable(init_image, dtype=tf.float32)
  # Create our optimizer
    opt = tf.train.AdamOptimizer(learning_rate=5, beta1=0.99, 
epsilon=1e-1)

  # For displaying intermediate images
  iter_count = 1

  # Store our best result
  best_loss, best_img = float('inf'), None

  # Create a nice config
  loss_weights = (style_weight, content_weight)
  cfg = {
      'model': model,
      'loss_weights': loss_weights,
      'init_image': init_image,
      'gram_style_features': gram_style_features,
      'content_features': content_features
  }
  # For displaying
  num_rows = 2
  num_cols = 5
  display_interval = num_iterations/(num_rows*num_cols)
  start_time = time.time()
  global_start = time.time()

  norm_means = np.array([103.939, 116.779, 123.68])
  min_vals = -norm_means
  max_vals = 255 - norm_means

  imgs = []
  for i in range(num_iterations):
    grads, all_loss = compute_grads(cfg)
    loss, style_score, content_score = all_loss
    opt.apply_gradients([(grads, init_image)])
    clipped = tf.clip_by_value(init_image, min_vals, max_vals)
    init_image.assign(clipped)
    end_time = time.time()

    if loss < best_loss:
      # Update best loss and best image from total loss.
      best_loss = loss
      best_img = deprocess_img(init_image.numpy())

    if i % display_interval== 0:
      start_time = time.time()

      # Use the .numpy() method to get the concrete numpy array
      plot_img = init_image.numpy()
      plot_img = deprocess_img(plot_img)
      imgs.append(plot_img)
      IPython.display.clear_output(wait=True)
      IPython.display.display_png(Image.fromarray(plot_img))
      print('Iteration: {}'.format(i))
      print('Total loss: {:.4e}, '
            'style loss: {:.4e}, '
            'content loss: {:.4e}, '
          'time: {:.4f}s'.format(loss, style_score, content_score,
time.time() - start_time))
  print('Total time: {:.4f}s'.format(time.time() - global_start))
  IPython.display.clear_output(wait=True)
  plt.figure(figsize=(14,4))
  for i,img in enumerate(imgs):
      plt.subplot(num_rows,num_cols,i+1)
```

```
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])

    return best_img, best_loss
```

The optimizer used in here is Adam optimizer. Adam optimizer is great to minimize the loss. Here, we do not work on updating weights, instead we decrease the loss; for that loss and gradients should be calculated [11].

6 Variational Autoencoders

Neural networks are great for classification tasks, but that does not mean they cannot do regression or clustering tasks. Any generative model needs some random numbers which looks similar to training data. But more extensively, models often require explorations on variation of data that we already have. Here, variational autoencoders works much better.

Generally, there is an encoder and a decoder in an autoencoder network that takes an input and splits them into small denser features which a decoder will combine to the original input. Since you're already familiar with convolution, you'll learn autoencoders without a problem. Like a ConvNet takes up a large image and condense it to smaller yet denser representation that feeds into a fully connected network, encoder takes the input and reduces that into a smaller representation using an encoding function. Autoencoder uses this and implants a decoding function which ConvNets cannot do.

Standard autoencoders are limited in reconstruction. The space where the inputs are converted into encoding vectors is called latent space. In standard autoencoders this latent space is not continuous and easy to interpolate. For example, MNIST data visualization in 2D latent space forms distinct clusters when we use a standard autoencoder, and when we decode, we might loss some of the data but not in a huge scale. But when we try to build a generative model out of it, we need to generate variations from a continuous latent space. If the space is not continuous, then the decoder will generate some random unrelated unrealistic output; this is because the decoder does not know how the latent space works.

Variational autoencoders are unique from traditional autoencoders in latent space design; this allows them to be more useful in generative models allowing random interpolation and sampling easily (Fig. 14).

Unlike vanilla encoders, this autoencoder takes the input to give not one but two vectors of size n: one is a mean vector, and another is a standard deviation vector. With that mean and standard deviation, we can generate samples having normal distribution

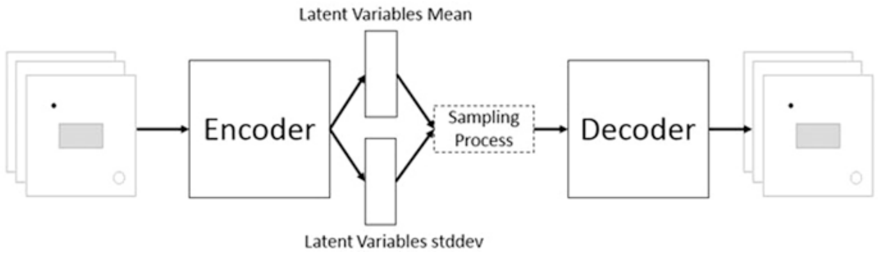


Fig. 14 Structure of variational autoencoder. (Reichstaller and Knapp [13]. Compressing Uniform Test Suites Using Variational Autoencoders. https://doi.org/10.1109/QRS-C.2017.128)

$$F(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

with μ and σ being mean and standard deviation for a random number X_i; the obtained encoding will then pass into the decoder. For example, let $\mu = [0.1, 1.2, 0.2, 0.8, ..]$ and $\sigma = [0.2, 0.5, 0.8, 1.3]$; random numbers can be generated from $[X_1{\sim}N(0.1, 0.2^2), X_2\ !\ N(1.2, 0.5^2),\ X_3{\sim}N(0.2, 0.8^2),\ X_4{\sim}N(0.8, 1.3^2), \ldots]$ resulting in a sample vector [0.28, 1.65, 0.95, 1.98, ..]. This resultant sample vector generation process is called stochastic generation. The mean and standard deviation are the same as that of the input for the stochastically generated samples.

The mean vector is the base for samples, and the standard deviation decides how much mean to be encoded; this space where mean and standard deviation control is called distribution. New samples added to the distribution will spread the space for more sample generation. The decoder on the other hand learns that samples are not generating from a single point, but they follow a distribution in the continuous latent space. The region of spread from a single point is local to that point only, and its exposure to the model varies for a certain degree, thus forming a smoothness in local space. This smooth local space of a set of samples is referred as a cluster.

To provide an intuition of variational autoencoders, let us consider an example with a huge dataset having different faces. Our aim here is to cluster similar images. Vanilla autoencoder will first encode each image in latent space as shown in Fig. 15.

As you can see, the image is described in single-valued latent attributes. Single-valued attributes are hard for detection purposes; instead it is preferable to have a range of values. For instance, let's say a picture of Charles Babbage was encoded with a smile value of 0.12 which is very distinct from 0.99 but still comes under smiling cluster. Variational autoencoder will provide a probability distribution for each latent attribute as shown in Fig. 16.

When decoding, we will take samples from each latent attribute to generate a sample vector and then pass that to the decoder. In this way a slight variation in the input will result in accurate output (reconstruction) [12].

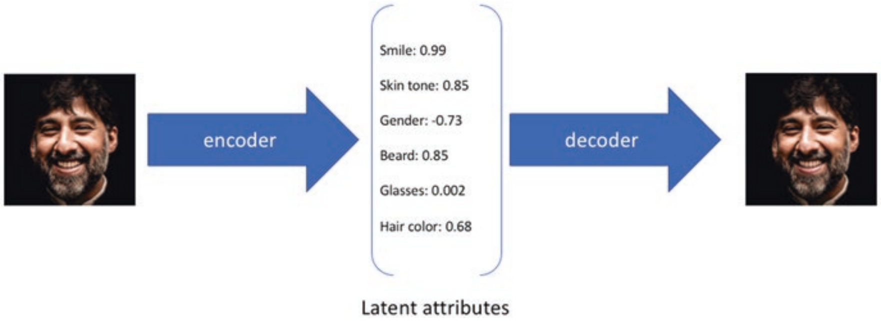


Fig. 15 Latent space representation of an image using vanilla autoencoders. (https://www.jeremyjordan.me/variational-autoencoders/)

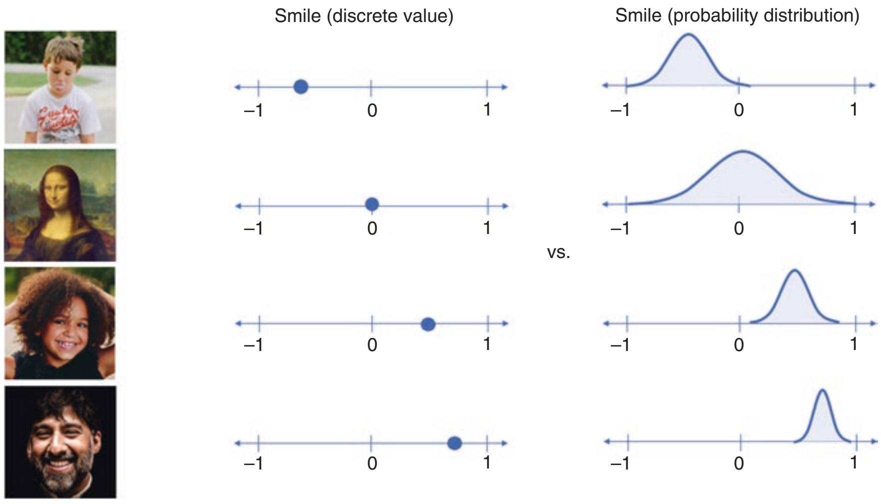


Fig. 16 Latent space representation of an image using variational autoencoders. (https://www.jeremyjordan.me/variational-autoencoders/)

Statistically, this method uses Bayesian theory. If z samples are to be generated from x observations, we will compute the probability of z with respect to x ($p(z|x)$):

$$p\left(z|x\right) = \frac{p\left(x|z\right)p\left(z\right)}{p\left(x\right)}$$

However, the value of $p(x)$ can only be estimated as its distribution is intractable:

$$p\left(x\right) = \int p\left(x|z\right)p\left(z\right)dz$$

Generally, $p(x|z)$ is approximated to another tractable distribution $q(x|z)$. To ensure that both $p(x|z)$ and $q(x|z)$ are similar, we must minimize the divergence between them. Here the divergence is measured to be KL divergence:

$$\min KL\left(q\left(z|x\right)\ \ p\left(z|x\right)\right)$$

This can be derived to

$$E_{q\left(z|x\right)} logp\left(x|z\right) - KL(q\left(z|x\right)||\left(z|x\right))\}$$

Here, the first term refers to likelihood reconstruction, and the second one is distribution of q similar to distribution p. This extends to the loss function having two terms, one dealing with the likelihood reconstruction errors and the other being the supporting function for distribution q:

$$L\left(x,\hat{x}\right) + \sum\_j\, KL\left(q\_j\left(z|x\right)\ \ p\left(z\right)\right)$$

The main advantage of variational encoders is the ability of learning smoothly. However, by generating samples from the latent space, we can produce a decoding network with the ability of generating new data identical to training data, thus making it a key inspiration for generative models.

7 DCGAN

While explaining about variational autoencoders, a word appeared many times, that is, generative models. Generative models can be described as an automatic model for clustering that discovers patterns by patterns in the input data and generates new data from it. A network involved with training such models is called generative adversarial networks (GANs) [14]. DCGAN is a popular GAN framework. It stands for deep convolution generative adversarial network [15]. Like the name suggests, it deals with convolution.

The network consists of many convolution layers with two types of convolution implementation. One with stride and another is transposed convolution. A Transposed convolution helps in upscaling the image which many researchers refer as Deconvolution, which according to me is wrong. Let's say a 5 × 5 image is convoluted with a 3 × 3 filter gives a 2 × 2 image and when we deconvolute that 2 × 2 image to get a 5 × 5 image by inversing the mathematical process. Transposed convolution is not like that; it adds some padding around it and then performs the familiar convolution function. A 2 × 2 image is padded in such a way that it forms a 3 × 3 space for each pixel like shown in Fig. 17 [16].

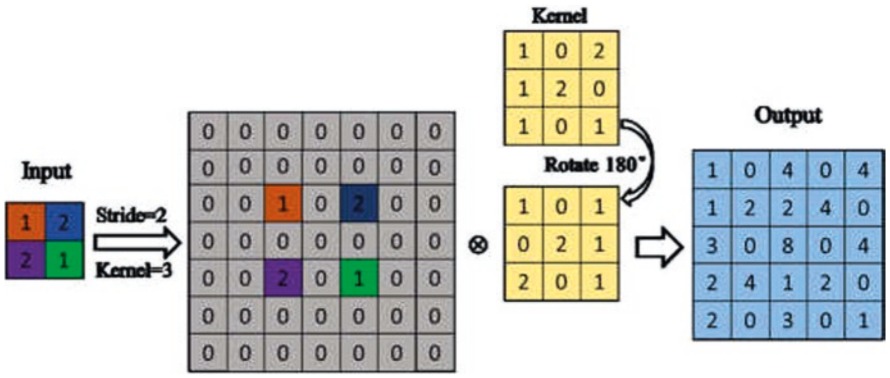


Fig. 17 Demonstration of transposed convolution. (Zhang et al. [16])

Basic structure of DCGAN is identical to any ConvNets, but every pooling layer is replaced with a convolution stride. A transposed convolution is added for upscaling. Let's see a GAN network in action:

```
def generator(img_shape, z_dim):
  model = Sequential()
  # Hidden layer
  model.add(Dense(128, input_dim = z_dim))
  # Leaky ReLU
  model.add(LeakyReLU(alpha=0.01))
  # Output layer with tanh activation
  model.add(Dense(28*28*1, activation='tanh'))
  model.add(Reshape(img_shape)
  z = Input(shape=(z_dim,))
  img = model(z)
  return Model(z, img)
```

A leakyReLU activation is used here for allowing a little gradient of negative values. As a result, it will allow small negative gradients during backpropagation. Now with reference from the above code, let's build a code for DCGAN [17].

```
def generator(img_shape, z_dim):
  model = Sequential()

  # Reshape input into 7x7x256 tensor via a fully connected layer
  model.add(Dense(256*7*7, input_dim = z_dim))
  model.add(Reshape((7,7,256))

 # Transposed convolution layer, from 7x7x256 into 14x14x128 tensor
  model.add(Conv2DTranspose(
              128, kernel_size = 3, strides = 2, padding='same'))

 #Batch normalization
  model.add(BatchNormalization())

  #Leaky ReLU
  model.add(LeakyReLU(alpha=0.01))

 # Transposed convolution layer, from 14x14x128 to 14x14x64 tensor
  model.add(Conv2DTranspose(
              64, kernel_size=3, strides=1, padding='same'))

  # Batch normalization
  model.add(BatchNormalization())

  # Leaky ReLU
  model.add(LeakyReLU(alpha=0.01))

  # Transposed convolution layer, from 14x14x64 to 28x28x1 tensor
  model.add(Conv2DTranspose(
               1, kernel_size = 3, strides = 2, padding='same'))

  # Tanh activation
  model.add(Activation('tanh'))
  z = Input(shape=(z_dim,))
  img = model(z)
  return Model(z, img)
```

The input for the above architecture is projected from a 100×1 generated sample that was upscaled to $1024 \times 4 \times 4$ tensor which is then passed through convolution until we get $64 \times 64 \times 3$ output as shown in Fig. 18 [18]. The simplicity of DCGAN is the main influence on its popularity. For starters DCGAN has been a great point [20, 21] towards generative models. Most of the GANs used these days are inspired by the DCGAN architecture [22–24].

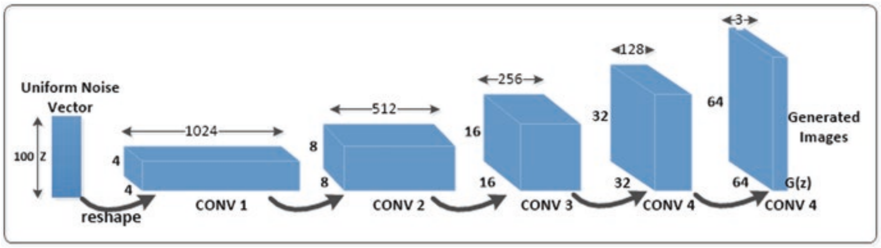


Fig. 18 DCGAN structure or architecture. (Shorten and Khoshgoftaar [18])

References

1. Yellapragada, B., Kolla, B.: Effective handwritten digit recognition using deep convolution neural network. Int. J. Adv. Trends Comput. Sci. Eng. **9** (2020). https://doi.org/10.30534/ijatcse/2020/66922020
2. Hidaka, A., Kurita, T.: Consecutive dimensionality reduction by canonical correlation analysis for visualization of convolutional neural networks. Proc. ISCIE Int. Symp. Stoch. Syst. Theory Appl., 160–167 (2017, 2017). https://doi.org/10.5687/sss.2017.160
3. van der Vaart Wouter, V., Lambers, K.: Learning to look at LiDAR: the use of R-CNN in the automated detection of archaeological objects in LiDAR data from the Netherlands. J. Comput. Appl. Archaeol. **2**, 31–40 (2019). https://doi.org/10.5334/jcaa.32
4. Hossain, Md. M., Talbert, D., Ghafoor, S., Kannan, R.: FAWCA: A Flexible-greedy Approach to find Well-tuned CNN Architecture for Image Recognition Problem. (2018)
5. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. Neural Inf. Process. Syst. **25** (2012). https://doi.org/10.1145/3065386
6. Hughes, D. & Correll, N.: Distributed Machine Learning in Materials that Couple Sensing, Actuation, Computation and Communication. (2016)
7. Yuan, X., Li, L., Wang, Y.: Nonlinear dynamic soft sensor modeling with supervised long short-term memory network. IEEE Trans. Ind. Inf., 3168–3176 (2019). https://doi.org/10.1109/TII.2019.2902129
8. Sua, Y., Kuo, C.: On extended long short-term memory and dependent bidirectional recurrent neural network. Neurocomputing. **356**, 151–161 (2018). https://doi.org/10.1016/j.neucom.2019.04.044
9. Naber, M., Hilger, M.: Wolfgang Einhäuser; animal detection and identification in natural scenes: image statistics and emotional valence. J. Vis. **12**(1), 25 (2012). https://doi.org/10.1167/12.1.25
10. Chen, J., Yang, G., Zhao, H., et al.: Audio style transfer using shallow convolutional networks and random filters. Multimed. Tools Appl. **79**, 15043–15057 (2020). https://doi.org/10.1007/s11042-020-08798-6
11. Gatys, L., Ecker, A., Bethge, M.: A neural algorithm of artistic style. arXiv, https://doi.org/10.1167/16.12.326. (2015)
12. Doersch, C.: (2016). Tutorial on Variational Autoencoders, arXiv - 1606.05908, https://arxiv.org/abs/1606.05908
13. Reichstaller, A., Knapp, A.: Compressing uniform test suites using Variational autoencoders. In: IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 435–440. IEEE (2017). https://doi.org/10.1109/QRS-C.2017.128
14. Nghiem, P., Phan, N.: Generative Adversarial Networks (2019). https://doi.org/10.13140/RG.2.2.31946.62401
15. Goodfellow, Ian. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks

16. Zhang, Y., Zhao, X., Liu, P.: Multi-point displacement monitoring based on full convolutional neural network and smartphone. IEEE Access. **7**, 139628–139634 (2019). https://doi.org/10.1109/ACCESS.2019.2943599
17. Jakub, L., Vladimir, B.: GANs in Action, Chapter 3 ,September 2019 ISBN 9781617295560
18. Shorten, C., Khoshgoftaar, T.: A survey on image data augmentation for deep learning. J. Big Data. **6** (2019). https://doi.org/10.1186/s40537-019-0197-0
19. Poots, J.; Tensorflow: Past the Basics. (2019)
20. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
21. JHA. A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
22. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
23. Prakash, K.B.,. Ruwali, A., Kanagachidambaresan, G.R.: Regression, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_4
24. Vadla, P.K., Ruwali, A., Lakshmi, M.V.P., Kanagachidambaresan, G.R.: Neural network, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_5

Potential Applications of Deep Learning in Bioinformatics Big Data Analysis

Jayakishan Meher

1 Introduction

The rate of growth of biological data in recent years seems to be doubled in every 15 months. Substantial amounts of sequence, biomedical image and signal are being collected for use in medical and healthcare research. Such omics big data must be analysed for transformation into useful knowledge in bioinformatics research fields and thus remains as a key problem in computational biology.

Machine learning models have been used to find useful information from omics data. These algorithms utilize training data to reveal patterns and determine predictions. Machine learning approaches deals with learning relationships from data which does not require to be predefined [1–3]. The traditional machine learning algorithms depend on feature data, and thus its performance is determined. The intensive prediction task in the field of genomics [4, 5] and proteomics [6] as well as metabolomics research [7] depends on these machine learning techniques.

Deep learning technique is the evolving generation of artificial intelligence and advancement of machine learning that has exhibited exceptional prediction performance recently on big data analytics in image processing, signal processing and sequence analysis [8–10]. Deep learning has been inherited from artificial neural network with advanced features [11–17]. Deep learning neural networks (DNNs) are prominent for their fitness in analysing high-dimensional data. Since biological data appear to be high-dimensional and complex, hence DNN methods are suitable for bioinformatics research. DNNs can determine unknown abstract patterns and correlations to recognize the characteristics of data [18, 19]. The DNNs in bioinformatics will help to code raw data and study features effectively. The goal of this

J. Meher (✉)
Centurion University of Technology and Management, Gajapati, Odisha, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_7

review is to summarize the vital concepts and the potential applications of deep learning in bioinformatics big data analysis.

2 Bioinformatics Big Datasets

The quantity and degree of growth of biological data is cumulative exponentially with the discovery of new and automated sequence methods. Bioinformatics has evolved with publicly available large databases along with a plenty number of bioinformatics tools. The availability of DNA in GenBank [20] and protein structures in Protein Data Bank (PDB) [21] has motivated many data scientists and researchers to effectively perform biomolecule big data analysis. Substantial volumes of biomedical data, including omics in the form of images, sequences and other types of signals, enable potential applications in biomedical and healthcare research. Progression in computational platform like machine intelligence and signal processing has motivated a new dimension for bimolecular data analysis.

The most widely used biological databases include genome and protein sequence databases which serve as repositories of primary source for experimental results. The most widely used biological databank resources on the WWW are the genomic, proteomic and microarray information [22, 23]. Sequence databases are applicable to both nucleic acid sequences like GenBank, EMBL-Bank and DDBJ and protein sequences like PDB, Gene3D, PIR etc.

The microarray databases consist of array of images representing genes that allow to perform analysis of thousands of genes simultaneously. These databases like GEO, Genset, ArrayExpress etc. contain data obtained from microarray-based experiments [24, 25]. Table 1 summarizes data repositories of several prominent databases in this field.

3 Concepts in Deep Neural Network

Deep learning has evolved as a branch of machine learning for analysing big data with advances of artificial intelligence (AI) [26]. Machine learning finds patterns from known datasets with a limitation in processing raw data. Deep learning has shown ability to learn complex features. The architecture of deep learning is composed of artificial neural networks of multiple non-linear layers, and these hierarchical forms of data can be revealed by growing stages of abstraction [27].

In the elementary structure of DNN, there are three layers such as an input layer, the intermediate many hidden layers and finally an output layer as shown in Fig. 1. The sum of the product of weight vector for each unit in the current layer with the signal produces the output expression. The output values can be calculated by applying the weighted sum through a non-linear function such as a sigmoid, rectified linear unit etc. [28].

Table 1 Summary of bioinformatics databases

Bioinformatics database	Descriptive features
EMBL (www.ebi.ac.uk/embl)	This maintains database for nucleotide sequences
GenBank (www.ncbi.nlm.nih.gov)	This provides database for nucleic acid sequences
DDBJ (www.ddbj.nig.acjp)	This is an archive of nucleotide sequence database
TAIR (www.arabidopsis.org)	It maintains database of genomic information of *Arabidopsis thaliana*
dbEST (www.ncbi.nlm.nih.gov/dbEST)	It contains database of expressed sequence tag
PIR (pir.georgetown.edu)	This site maintains database of sequences and the functional information of protein
Swiss-Prot (www.expasy.org/sprot)	It is a source for protein sequences and functional information
UniProt (www.pir.uniprot.org)	It is a repository for protein sequence and function
PDB (www.rcsb.org/pdb)	It is an archive of 3D structures of protein structure and biological macromolecules
NDB (ndbserver.rutgers.edu)	It maintains database of nucleic acid structural information
SMD (genome-www5.stanford.edu)	This site maintains database of microarray data
Yale Microarray Database (www.med.yale.edu/microarray)	It maintains database of microarray data

The error in the training is minimized through the backward pass using the chain rule to back propagate error signals. The gradients are computed for all weights in the network [29]. The optimization algorithms are used based on stochastic gradient descent (SGD) for updating the weight parameters [30]. In the training, regularization performs strategies to eliminate overfitting and attain better generalization performance [31–33].

Depending on the nature of datasets and objectives, there are numerous forms of deep learning used like deep neural networks (DNNs) [34–38], convolutional neural networks (CNNs) [39–41], recurrent neural networks (RNNs) [42–45] etc. The most important advancements have been in speech and image detection [46–52], natural language processing [53, 54] and language conversion [55, 56]. Currently deep learning is increasingly used in bioinformatics research applications.

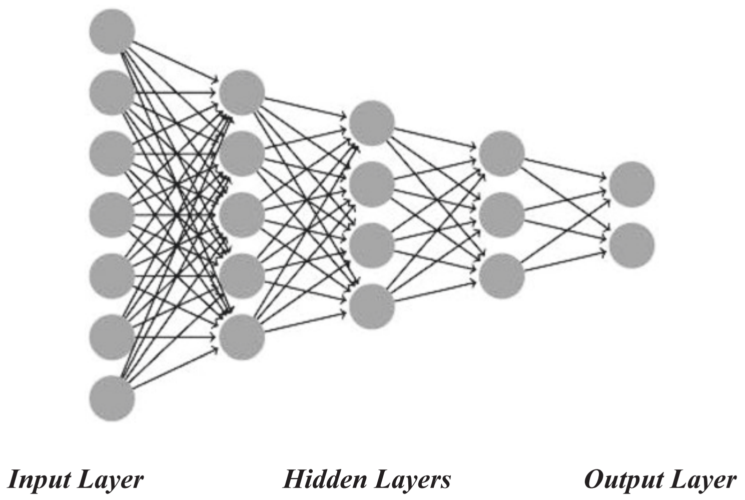


Fig. 1 Basic structure of DNNs

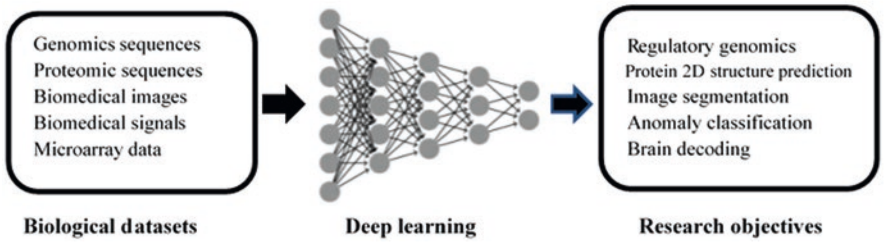


Fig. 2 Deep learning in bioinformatics research objectives

4 Applications of DNN in Bioinformatics

Deep learning has several potential applications in bioinformatics research fields to analyse big data as shown in Fig. 2. The application of deep learning architectures such as DNN, CNN and RNN in bioinformatics domain such as omics research, biomedical image processing and biomedical signal processing are being presented here.

4.1 Deep Learning for Omics Research

Deep learning has been adopted extensively in omics study such as genome, proteome and transcriptome data in bioinformatics and offers the most common raw biological sequence data such as strings of DNA, RNA and amino acids to be used as inputs in deep learning. Besides this, physicochemical properties [57, 58], one-dimensional structural properties [59, 60] and position-specific scoring matrices (PSSM) [61] extracted features from sequences are used as inputs for deep learning algorithms for handling high-dimensional biological data. Protein contact maps representing the distances of amino acid pairs in 3D structure of protein and microarray image gene expression are also utilized (Table 2). Xiong et al. [62] applied DNN effectively to predict splicing activity, and Alipanahi et al. [63] performed specificities of DNA- and RNA-binding proteins or epigenetic marks.

4.2 *Deep Learning for Protein Structure*

DNNs have been extensively applied in proteomic research [64–67] for protein secondary structure prediction as prediction in 3D space is complex. Heffernan et al. [65] used SAE to protein to predict successfully protein secondary structure and torsion angle. Similarly, Spencer et al. [66] predicted protein secondary structure by utilizing DBN to amino acid sequences with features like PSSM and Atchley factors. DNNs have also been used in gene expression regulation [67–73]. Lee et al. [68] applied DBN in splice junction prediction in understanding gene expression [74]. Asgari et al. [75] adopted the skip-gram model, a variant of MLP, to efficiently learn a distributed form of sequences in protein family classification.

Table 2 Deep learning applications in bioinformatics

Bioinformatics research fields	Research problems	Input datasets	Deep learning techniques
Genomics and proteomics	Protein secondary structure prediction Protein functional classification Anomaly classification Gene expression data regulation	Sequencing data: DNA, RNA, amino acid seq) Features: PSSM, physicochemical properties One-dimensional structural property Contact map Microarray gene expression	DNN CNN RNN
Biomedical image processing	Anomaly classification Segmentation Recognition Brain decoding	Radiographic image Magnetic resonance image Histopathology image Retinal image Positron emission tomography	DNN CNN
Biomedical signal processing	Brain decoding Anomaly classification	*ECoG* *ECG* *EMG* *EOG* EEG	DNN CNN RNN

4.3 Deep Learning for Biomedical Image Processing

The applications of deep learning have been extended to biomedical image processing field where biomedical images such as MRI, positron emission tomography (PET), radiographic imaging and histopathology imaging have been used for analysis in anomaly classification [76–78], recognition [79, 80], segmentation [81] and brain decoding [82, 83]. Plis et al. [84] classified schizophrenia patients from brain MRIs using DBN, whereas Xu et al. [85] used SAE to detect cell nuclei from histopathology images.

4.4 Biomedical Signal Processing

Another promising avenue of deep learning is the biomedical signal processing which uses signal from ECG, EEG, electrocorticography (ECoG), electrooculography (EOG) and electromyography (EMG) for analysis. The recorded signals are generally found to be noisy; hence, these signals are preprocessed by decomposing into wavelet components for input to deep learning algorithms. An et al. [86] utilized DBN to the frequency components of EEG signals to classify left- and right-hand motor imagery skills in brain decoding. Besides this Jia et al. [87] utilized DBN, and Jirayucharoensak et al. [88] utilized SAE for effective emotion classification.

4.5 Multimodal Deep Learning

Multimodal deep learning is a promising way in the advancement of biological research which explores information from multiple resources to be integrated like in addition to omics data, images and signals; other forms such as X-ray, CT, MRI and PET are also obtainable. Suk et al. [89] analysed Alzheimer's disease classification with the help of cerebrospinal fluid and brain images available in both MRI and PET scan. Soleymani et al. [90] led an emotion detection with the help of face image and EEG signal [91–93].

5 Conclusions

Deep learning techniques are the most promising machine learning tools for analysis in bioinformatics to derive hidden knowledge from big data. An extensive review of bioinformatics research using deep learning techniques in terms of heterogeneous input data and domain research objectives has been presented. It is seen that deep learning architectures such as DNN, CNN and RNN have been utilized extensively for bioinformatics research domain such as omics, biomedical image processing and biomedical signal processing to produce promising results. Applications of deep learning are key to success to advance bioinformatics in future research.

References

1. Hastie, T., Tibshirani, R., Friedman, J., Franklin, J.: The elements of statistical learning: data mining, inference and prediction. Math. Intell. **27**, 83–85 (2005)
2. Murphy, K.P.: Machine Learning: a Probabilistic Perspective. MIT Press, Cambridge (2012)
3. Michalski, R.S., Carbonell, J.G., Mitchell, T.M.: Machine Learning: an Artificial Intelligence Approach. Springer Science & Business Media, Berlin\Heidelberg (2013)
4. Libbrecht, M.W., Noble, W.S.: Machine learning applications in genetics and genomics. Nat. Rev. Genet. **16**, 321–332 (2015)
5. Märtens, K., Hallin, J., Warringer, J., Liti, G., Parts, L.: Predicting quantitative traits from genome and phenome with near perfect accuracy. Nat. Commun. **7**, 11512 (2016)
6. Swan, A.L., Mobasheri, A., Allaway, D., Liddell, S., Bacardit, J.: Application of machine learning to proteomics data: classification and biomarker identification in postgenomics biology. OMICS. **17**, 595–610 (2013)
7. Kell, D.B.: Metabolomics, machine learning and modelling: towards an understanding of the language of cells. Biochem. Soc. Trans. **33**, 520–524 (2005)
8. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science. **313**, 504–507 (2006). https://doi.org/10.1126/science.1127647
9. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature. **521**, 436 (2015). https://doi.org/10.1038/nature14539
10. Nussinov, R.: Advancements and challenges in computational biology. PLoSComput. Biol. **11**, e1004053 (2015). https://doi.org/10.1371/journal.pcbi.1004053
11. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., et al.: Human-level control through deep reinforcement learning. Nature. **518**, 529–533 (2015). https://doi.org/10.1038/nature14236
12. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85 (2015). https://doi.org/10.1016/j.neunet.2014.09.003
13. Mamoshina, P., Vieira, A., Putin, E., Zhavoronkov, A.: Applications of deep learning in biomedicine. Mol. Pharm. **13**, 1445–1454 (2016). https://doi.org/10.1021/acs.molpharmaceut.5b00982
14. Bengio, Y., LeCun, Y.: Scaling learning algorithms toward AI. In: Bottou, L., Chapelle, O., DeCoste, D., Weston, J. (eds.) Large-Scale Kernel Machines. The MIT Press, Cambridge, MA (2007)
15. Alipanahi, B., Delong, A., Weirauch, M.T., Frey, B.J.: Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. Nat. Biotechnol. **33**, 831–838 (2015). https://doi.org/10.1038/nbt.3300
16. Libbrecht, M.W., Noble, W.S.: Machine learning applications in genetics and genomics. Nat. Rev. Genet. **16**, 321–322 (2015). https://doi.org/10.1038/nrg3920
17. Zhang, S., Zhou, J., Hu, H., Gong, H., Chen, L., Cheng, C., Zeng, J.: A deep learning framework for modeling structural features of RNA-binding protein targets. Nucleic Acids Res. **44**, e32 (2016). https://doi.org/10.1093/nar/gkv1025
18. Esteva, A., Kuprel, B., Novoa, R.A., Ko, J., Swetter, S.M., Blau, H.M., Thrun, S.: Dermatologist-level classification of skin cancer with deep neural networks. Nature. **542**, 115–118 (2017). https://doi.org/10.1038/nature21056

19. Ching, T., Himmelstein, D.S., Beaulieu-Jones, B.K., Kalinin, A.A., Do, B.T., Way, G.P., et al.: Opportunities and obstacles for deep learning in biology and medicine. J. R. Soc. Interface. **15**, 20170387 (2018). https://doi.org/10.1098/rsif.2017.0387
20. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Rapp, B.A., Wheeler, D.L.: Nuleic Acid Res. **28**, 15–18 (2000)
21. Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissib, H., Shindyalov, J.N., Bourne, P.E.: Nucleic Acids Res. **28**, 235–242 (2000)
22. Barker, W.C., Garavelli, J.S., PB, M.G., Marzee, C.R., Orcutt, B.C., Srinivarsarao, G.Y., Yeh, L.S., Mewes, H.W., Pfeiffer, F., et al.: Nucleic Acid Res. **27**, 39–43 (1999)
23. Bairoch, A., Apweiler, R.: Nucleic Acid Res. **27**, 44–48 (1999)
24. Schena, M., Shalon, D., Davis, R.W., Brown, P.O.: Quantitative monitoring of gene expression patterns with a complementary DNA microarray. Science. **270**, 467–470 (1995)
25. Ball, C.A., Awad, I.A., Demeter, J., et al.: The Stanford microarray database accommodates additional microarray platforms and data formats. Nucleic Acids Res. **33**(Database issue), D580–D582 (2005)
26. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature. **521**(7553), 436–444 (2015)
27. LeCun, Y., Ranzato, M.: Deep learning tutorial. In: Tutorials in International Conference on Machine Learning (ICML'13). Citeseer (2013)
28. Nair, V., Hinton, G.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 807–814 (2010)
29. Hecht-Nielsen, R.: Theory of the backpropagation neural network. In: International Joint Conference on Neural Networks, 1989. IJCNN, pp. 593–605. IEEE, Washington, DC, USA (1989)
30. Bottou, L.: Stochastic gradient learning in neural networks. Proc. Neuro-. Nımes. **91**(8), 12 (1991)
31. Moody, J., Hanson, S., Krogh, A., et al.: A simple weight decay can improve generalization. Adv. Neural Inf. Proces. Syst. **4**, 950–957 (1995)
32. Srivastava, N., Hinton, G., Krizhevsky, A., et al.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)
33. Baldi, P., Sadowski, P.J.: Understanding dropout. In: Advances in Neural Information Processing Systems, NeurIPS Proceedings. pp. 2814–2822 (2013)
34. Svozil, D., Kvasnicka, V., Pospichal, J.: Introduction to multilayer feed-forward neural networks. Chemom. Intell. Lab. Syst. **39**(1), 43–62 (1997)
35. Vincent, P., Larochelle, H., Bengio, Y., et al.: Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th International Conference on Machine Learning, pp. 1096–1103. ACM, New York (2008)
36. Vincent, P., Larochelle, H., Lajoie, I., et al.: Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**, 3371–3408 (2010)
37. Hinton, G., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
38. Hinton, G., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science. **313**(5786), 504–507 (2006)
39. LeCun, Y., Boser, B., Denker, J.S., et al.: Handwritten digit recognition with a back-propagation network. In: Advances in Neural Information Processing Systems, Citeseer (1990)
40. Lawrence, S., Giles, C.L., Tsoi, A.C., et al.: Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Netw. **8**(1), 98–113 (1997)
41. Krizhevsky, A., Sutskever, I., Hinton, G.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
42. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Comput. **1**(2), 270–280 (1989)

43. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
44. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
45. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. Neural Comput. **12**(10), 2451–2471 (2000)
46. Farabet, C., Couprie, C., Najman, L., et al.: Learning hierarchical features for scene labeling. IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1915–1929 (2013)
47. Szegedy, C., Liu, W., Jia, Y., et al.: Going deeper with convolutions. arXiv Preprint arXiv. **1409**, 4842 (2014)
48. Tompson, J.J., Jain, A., LeCun, Y., et al.: Joint training of a convolutional network and a graphical model for human pose estimation. In: Advances in Neural Information Processing Systems, pp. 1799–1807 (2014)
49. Liu, N., Han, J., Zhang, D., et al.: Predicting eye fixations using convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 362–370 (2015)
50. Hinton, G., Deng, L., Yu, D., 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 (2012)
51. Sainath, T.N., Mohamed, A.-R., Kingsbury, B., et al.: Deep convolutional neural networks for LVCSR. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 8614–8618. IEEE, New York (2013)
52. Chorowski, J.K., Bahdanau, D., Serdyuk, D., et al.: Attention-based models for speech recognition. Adv. Neural Inf. Process. Syst. **2015**, 577–585 (2015)
53. Kiros, R., Zhu, Y., Salakhutdinov, R.R., et al.: Skip-thought vectors. Adv. Neural Inf. Proces. Syst. **2015**, 3276–3284 (2015)
54. Li, J., Luong, M.-T., Jurafsky, D.: A hierarchical neural autoencoder for paragraphs and documents. arXiv Preprint arXiv, 1506.01057 (2015)
55. Luong, M.-T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. arXiv Preprint arXiv, 1508.04025 (2015)
56. Cho, K., Van Merriënboer, B., Gulcehre, C., et al.: Learning phrase representations using RNN encoder–decoder for statistical machine translation. arXiv Preprint arXiv, 1406.1078 (2014)
57. Ponomarenko, J.V., Ponomarenko, M.P., Frolov, A.S., et al.: Conformational and physicochemical DNA features specific for transcription factor binding sites. Bioinformatics. **15**(7), 654–668 (1999)
58. Cai, Y.-D., Lin, S.L.: Support vector machines for predicting rRNA-, RNA-, and DNA-binding proteins from amino acid sequence. BiochimBiophys Acta (BBA) – proteins. Proteomics. **1648**(1), 127–133 (2003)
59. Branden, C.I.: Introduction to Protein Structure. Garland Science, New York (1999)
60. Richardson, J.S.: The anatomy and taxonomy of protein structure. Adv. Protein Chem. **34**, 167–339 (1981)
61. Jones, D.T.: Protein secondary structure prediction based on position-specific scoring matrices. J. Mol. Biol. **292**(2), 195–202 (1999)
62. Xiong, H.Y., Alipanahi, B., Lee, L.J., Bretschneider, H., Merico, D., Yuen, R.K.C., Hua, Y., Gueroussov, S., Najafabadi, H.S., Hughes, T.R., Morris, Q., Barash, Y., Krainer, A.R., Jojic, N., Scherer, S.W., Blencowe, B.J., Frey, B.J.: The human splicing code reveals new insights into the genetic determinants of disease. Science. **347**, 1254806 (2015)
63. Alipanahi, B., Delong, A., Weirauch, M.T., Frey, B.J.: Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. Nat. Biotechnol. **33**, 831–838 (2015)
64. Lyons, J., Dehzangi, A., Heffernan, R., et al.: Predicting backbone Ca angles and dihedrals from protein sequences by stacked sparse auto-encoder deep neural network. J. Comput. Chem. **35**(28), 2040–2046 (2014)

65. Heffernan, R., Paliwal, K., Lyons, J., et al.: Improving prediction of secondary structure, local backbone angles, and solvent accessible surface area of proteins by iterative deep learning. Sci. Rep. **5**, 11476 (2015)
66. Spencer, M., Eickholt, J., Cheng, J.: A deep learning network approach to ab initio protein secondary structure prediction. IEEE/ACM Trans. Comput. Biol. Bioinform. **12**(1), 103–112 (2015)
67. Nguyen, S.P., Shang, Y., Xu, D.: DL-PRO: a novel deep learning method for protein model quality assessment. In: 2014 International Joint Conference on Neural Networks (IJCNN), pp. 2071–2078. IEEE, New York (2014)
68. Leung, M.K., Xiong, H.Y., Lee, L.J., et al.: Deep learning of the tissue-regulated splicing code. Bioinformatics. **30**(12), i121–i129 (2014)
69. Lee, T., Yoon, S.: Boosted categorical restricted boltzmann machine for computational prediction of splice junctions. In: International Conference on Machine Learning, Lille, France, pp. 2483–2492 (2015)
70. Zhang, S., Zhou, J., Hu, H., et al.: A deep learning framework for modeling structural features of RNA-binding protein targets. Nucleic Acids Res. **44**(4), e32 (2015)
71. Chen, Y., Li, Y., Narayan, R., et al.: Gene expression inference with deep learning. Bioinformatics. **32**(12), 1832–1839 (2016)
72. Li, Y., Shi, W., Wasserman, W.W.: Genome-wide prediction of cis-regulatory regions using supervised deep learning methods. bioRxiv, 041616 (2016)
73. Liu, F., Ren, C., Li, H., et al.: De novo identification of replication timing domains in the human genome by deep learning. Bioinformatics. **32**, 641–649 (2015)
74. Nilsen, T.W., Graveley, B.R.: Expansion of the eukaryotic proteome by alternative splicing. Nature. **463**(7280), 457–463 (2010)
75. Asgari, E., Mofrad, M.R.: Continuous distributed representation of biological sequences for deep proteomics and genomics. PLoS One. **10**(11), e0141287 (2015)
76. Plis, S.M., Hjelm, D.R., Salakhutdinov, R., et al.: Deep learning for neuroimaging: a validation study. Front. Neurosci. **8**, 229 (2014)
77. Hua, K.-L., Hsu, C.-H., Hidayati, S.C., et al.: Computer-aided classification of lung nodules on computed tomography images via deep learning technique. Onco. Targets. Ther. **8**, 2015–2022 (2015)
78. Suk, H.-I., Shen, D.: Deep learning-based feature representation for AD/MCI classification. In: Medical Image Computing and Computer-Assisted Intervention – MICCAI, vol. 2013, pp. 583–590. Springer, New York (2013)
79. Xu, J., Xiang, L., Liu, Q., et al.: Stacked Sparse Autoencoder (SSAE) for nuclei detection on breast cancer histopathology images. IEEE Trans. Med. Imaging. **35**(1), 119–130 (2015)
80. Chen, C.L., Mahjoubfar, A., Tai, L.-C., et al.: Deep learning in label-free cell classification. Sci. Rep. **6**, 21471 (2016)
81. Li, Q., Feng, B., Xie, L., et al.: A cross-modality learning approach for vessel segmentation in retinal images. IEEE Trans. Med. Imaging. **35**(1), 109–108 (2015)
82. Gerven, M.A.V., De Lange, F.P., Heskes, T.: Neural decoding with hierarchical generative models. Neural Comput. **22**(12), 3127–3142 (2010)
83. Koyamada, S., Shikauchi, Y., Nakae, K., et al.: Deep learning of fMRI big data: a novel approach to subject-transfer decoding. arXiv Preprint arXiv, 1502.00093 (2015)
84. Plis, S.M., Hjelm, D.R., Salakhutdinov, R., et al.: Deep learning for neuroimaging: a validation study. Front. Neurosci. **8**, 229 (2014)
85. Xu, J., Xiang, L., Liu, Q., et al.: Stacked Sparse Autoencoder (SSAE) for nuclei detection on breast cancer histopathology images. IEEE Trans. Med. Imaging. **35**(1), 119–130 (2015)
86. An, X., Kuang, D., Guo, X., et al.: A deep learning method for classification of EEG data based on motor imagery. In: Intelligent Computing in Bioinformatics, pp. 203–210. Springer, Heidelberg (2014)

87. Jia, X., Li, K., Li, X., et al.: A novel semi-supervised deep learning framework for affective state recognition on EEG signals. In: 2014 IEEE International Conference on Bioinformatics and Bioengineering (BIBE), pp. 30–37. IEEE, New York (2014)
88. Jirayucharoensak, S., Pan-Ngum, S., Israsena, P.: EEG-based emotion recognition using deep learning network with principal component based covariate shift adaptation. Sci. World J. **2014**, 627892 (2014). https://doi.org/10.1155/2014/627892
89. Suk, H.-I., Shen, D.: Deep learning-based feature representation for AD/MCI classification. In: Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013, pp. 583–590. Springer, New York (2013)
90. Soleymani, M., Asghari-Esfeden, S., Pantic, M., et al.: Continuous emotion detection using EEG signals and facial expressions. In: 2014 IEEE International Conference on Multimedia and Expo (ICME), pp. 1–6. IEEE, New York (2014)
91. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
92. JHA, A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
93. Bharadwaj, Prakash, K.B., Kanagachidambaresan, G.R.: Kernel support vector machine, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_11

Dynamic Mapping and Visualizing Dengue Incidences in Malaysia Using Machine Learning Techniques

Nirbhay Mathur, Vijanth S. Asirvadam, Sarat C. Dass, and Balvinder Singh Gill

1 Introduction

Dengue is a life-threatening disease that gets evolved from the human body exposed to a bite of female mosquito belonging to *Aedes* species [1]. Dengue fever can be observed on the human body when *Aedes albopictus* bite and spread the dengue virus. Dengue fever was declared as a vector-borne disease that have symptoms of the arthropod-borne virus [2]. This disease has become one of the major concerns for public health. As per the World Health Organization (WHO), dengue cases are unreported and misclassified for dengue fever. Looking at figures, about 390 million dengue cases are estimated per year from which 96 million are reported clinically [3]. In the late 1870s (1870–1873), the first case of dengue, which was later declared as "pandemic," was reported from the coastal area of East Africa, Arabian Coast, and Port Said City [4–6].

Dengue-spreading mosquito belongs to the special breed of the Flaviviridae family. Dengue spreading mosquitoes are named as *Aedes aegypti* and *Aedes*

N. Mathur (✉) · V. S. Asirvadam
Research Scientist, Department of Electrical and Electronics Engineering, Universiti Teknologi PETRONAS, Seri Iskander, Perak, Malaysia
e-mail: nirbhay.mathur@utp.edu.my; vijanth_sagayan@utp.edu.my

S. C. Dass
Department of Mathematical and Computer Sciences, Heriot-Watt University, Putrajaya, Malaysia
e-mail: s.dass@hw.ac.uk

B. S. Gill
Disease Control Division, Ministry of Health, Malaysia, Putrajaya, Malaysia
e-mail: drbsgill@moh.gov.my

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_8

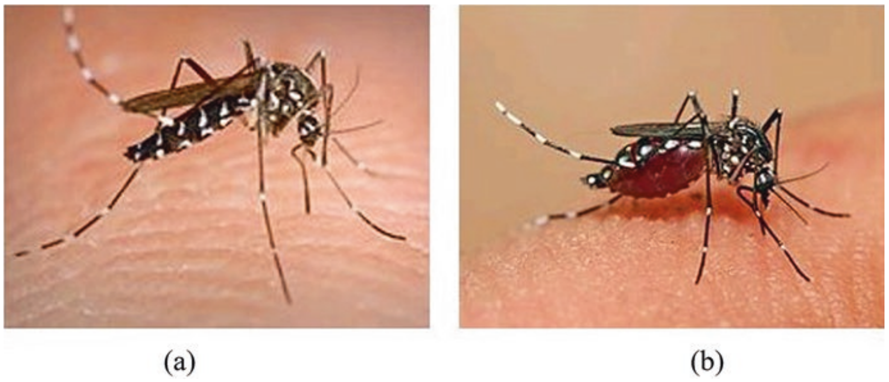


Fig. 1 Flaviviridae family mos. (**a**) *Aedes aegypti* female mosquito, (**b**) *Aedes albopictus* male mosquito [7]

albopictus, among which female mosquito is the dengue virus carries which is *Aedes albopictus*, as shown in Fig. 1.

The occurrence of dengue depends on some factors, but the major parameter is climate conditions such as rainfall, temperature, wind, and humidity. Among the mentioned variables, rainfall is considered as the most important factor for the spread of dengue virus. It has been noted that with high rainfall, with a time lag of 5–7 days, dengue fever cases are reported more. As after the rain, the rain water get collected to some junk places and get stored in unused containers such as old tires, tank, broken plastic containers, broken tanks [8, 9].

Mostly female mosquitoes of *Aedes aegypti* specie lay eggs in the inner surface of containers, on the upper layer of stored water. It had been recorded that a female mosquito can lay up to 100–250 eggs at each fertility time. Once the eggs are produced on the water surface of containers, it will take 7–9 days for them to develop and become a larva which is the first stage of development of the dengue virus carrier. Another four predevelopment stages are required to reach the second stage; this stage needs 25 °C for the larva to evolve. Another 2–3 days are required at the second stage to develop as a pupa, which can also be considered as a semi-developed stage of the mosquito. 2 more days are required to become a complete adult mosquito. *Aedes aegypti* wait to dry up their wings, and once it's done, they are ready to fly and spread dengue virus [10].

Once a mosquito is ready to bite the human body, the human becomes the source of infection and carries the virus with multipliers in spreading cases. After the first symptoms have appeared in an infected human, he/she can transmit infector for 4–5 days, with a maximum limit of 12 days. DENV has four different serotypes: DENV1, DENV2, DENV3, and DENV4. Dengue fever (DF) is confirmed if the patient is having fever and one of the following activities is present in them: headache, myalgia, hemorrhagic manifestations, rash, and leukopenia (Fig. 2) [11].

Approximately 5% of people diagnosed with dengue have more severe illness, and 1% has severe life-threatening infection [12]. Research in Malaysia has revealed

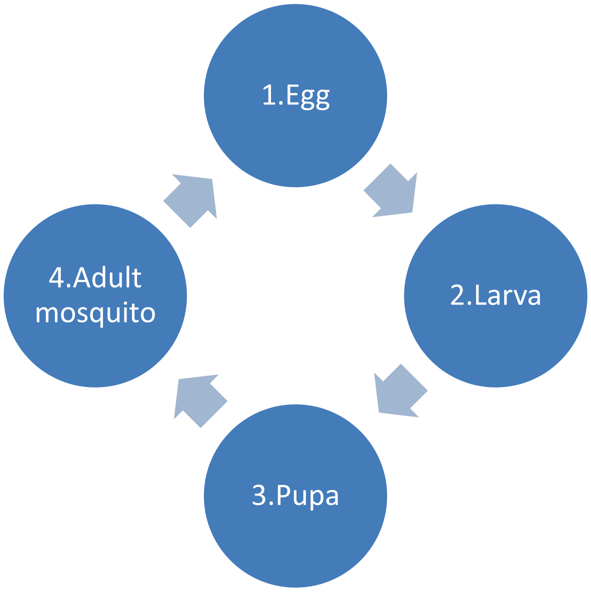


Fig. 2 *Aedes aegypti* female mosquito life cycle

an extensive jungle dengue transmission cycle involving canopy-feeding monkeys and *Aedes niveus*, a species that feeds on both monkey and humans [13]. Dengue infection is transmitted by the urban vectors *Aedes aegypti* and *Aedes albopictus* [14]. *Aedes aegypti* is predominantly a coastal species on large continents and is usually found in tropical regions. The presence of the virus is also found in countries such as Australia, the United States, and Brazil [15].

In 1902, the first outbreak of dengue virus was notified in Penang in November–December 1901 [16]. After the first outbreak, many other cases were recorded in urban developing areas of Penang and Kuala Lumpur [17–20]. By the 1960s, dengue was declared as endemic in Malaysia, and later in 1962, the first laboratory-confirmed dengue fever was reported in Penang [17]. Many research had been performed and still being performed in the present time. A research was conducted in which microclimate was the main parameter of the study, impacting dengue by varying associated latitude greater by 13 degrees in the north and south of the equator [21–23]. A study was found in which weather was a major parameter to explore the dengue incidences in Malaysia by calculating the density of occurred dengue cases [24]. Few other studies discussess the influence of temperature and another climate parameter such as the speed of the wind will help to find the lag time between the dengue outbreak and rainfall [25, 26]. In spite of all the conducted research, a visualization factor was still missing, and also a real-time mapping for predicted incidences was not found.

The main motivation of this research is to develop a spatial mapping for dengue incidences on real-time location using GIS. This study also predicts the vulnerability mapping for dengue incidences in the state of Malaysia. This research article is structured in the following sections:

Section 2: This section will discuss the overview of the related work done and other achievements acquired for dengue spread. This section will also talk about dengue virus spread in Malaysia and how climatic conditions are making it possible. Also, all predicting models will be summarized to understand the aspect from the predicting point of view and what tools can be used to develop the dengue spatial mapping that will be mentioned.

Section 3: This section will give a detailed introduction to the methodology used in this research. The mapping technique and machine learning predictive models will be discussed. This section will also explain the clustering technique such as the use of *k*-means to predict dengue cases and optimization techniques such as K-NN, and to validate the results expectation-maximization (EM) algorithm will be discussed.

Section 4: This section will demonstrate the results based on prediction and classification for spatial-temporal mapping of dengue incidences. The dynamic mapping will be represented to understand the trend and behavior of dengue incidences, which will help us to understand the most vulnerable area for dengue incidences.

Section 5: Last but not least, this section will summarize the results based on the proposed methodology. Here, a detailed discussion will be done by the author to explain the work proposed and the useful findings and future work recommendations.

2 Background

2.1 Dengue in Malaysia

Dengue fever (DF) has already become a global disease, which is spreading in all corners of the world. The spread can be seen in Fig. 3, where most of the developed countries are suffering from dengue virus.

As per WHO overall amount spend on dengue epidemics has been increasing in every interval; in some countries such as in Asia and America, they normally spend US $828 million and still counting [27]. As per the data, the total of cases recorded from 2008 to 2010 is about 136,992, which is the highest recorded over the century [28]. Malaysia was second highest in recording dengue incidences after Lao People's Democratic Republic, as shown in Table 1.

Malaysia was reported with 91.6% of dengue seropositive on the high cohort age of 35–74 years among the nation [29]. From 2010 to 2013, dengue incidences increase randomly from 16% to 0.62% [30, 31]. In 2010, a study was conducted at Negeri Sembilan, which recorded 1,466 dengue incidences; the youngest age affected was 8 months old, and the oldest age recorded was 89 years old, since the

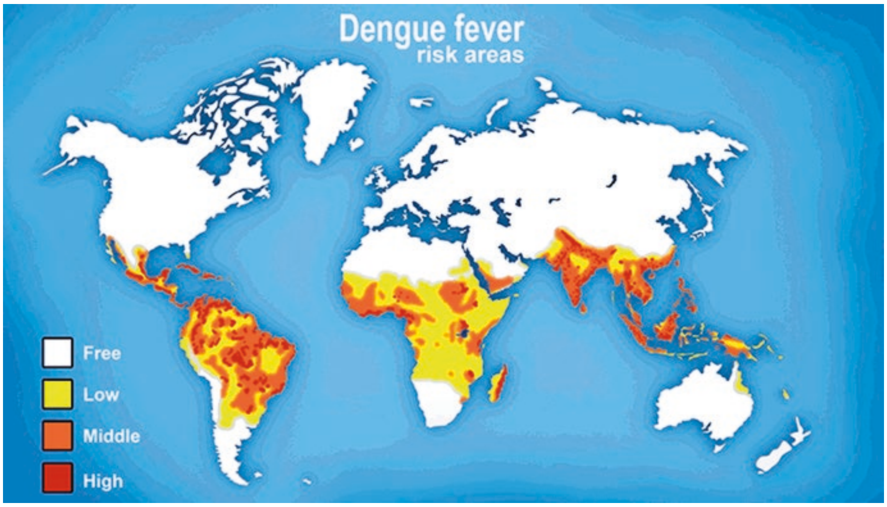


Fig. 3 Risk of dengue spread globally. (Source: Harvard T.H. Chan School of Public Health)

study concluded that the mean age of dengue occurrence in Malaysia was between 32.2 and 15.8 years [32]. This research also discussed about the ethnic groups of Malaysia, in which Malays were in the majority to have dengue incidences followed by Chinese and lest were Indian, ratio was about 4.1:1.5:1 and also this research tell about the gender to be more vulnerable to dengue cases were recorded as 1.4:1.0 where males are more affected than females [9].

As per WHO the vaccination of dengue virus is under clinical trials, and hence it was instructed to control this endemic by using vector control method. Malaysia is also practicing vector control methods such as environmental management, adulticiding, larviciding, and integrating control. Malaysia is also practicing fogging of chemicals to kill dengue mosquitoes in the area of registered cases [33]. Communication for behavioral impact (COMBI) is also a good initiative from Malaysia to control dengue spread, by searching and destroying dengue virus [34].

To control this endemic panic, the Malaysian Government had designed some strategic plans which were implemented in 2009–2013, as shown in Fig. 4.

These plans were divided into seven stages as follows: stage 1 is to conduct the surveillance and design the system which can update on the dengue incidences. In stage 2 an integrated vector management system will be performed. In stage 3 a dengue management will be involved to confirm the case for dengue. Once the confirmation is done for dengue cases, stage 4 will be active to do all the necessary communication and to perform social mobilization. In stage 5 the dengue outbreak team will be involved to give their response and to observe the control of spread. Once a case is understood by the outbreak team, all required data will be supplied to research team where a dengue research will be performed on the mentioned parameters. The last stage will be to implement the plan for a particular area.

Table 1 Number of dengue incidences registered in Asia and Pacific subregion

Countries/territories†	Cases	Notification (per 100 000)	Deaths	Case fatality rate (%)	Population (×1000)
Asia subregion					
Brunei Darussalam	25	6.16	0	0	406
Cambodia	15,980	119.29	73	0.46	13,396
China	124	001	0	0	1 370 537
Hong Kong (China)	30	0.42	0	0	7068
Japan	104	0.08	0	0	128,056
Republic of Korea	72	0.15	0	0	48,875
Lao People's Democratic Republic	3905	63.72	7	0.18	6128
Macao (China)	3	0.54	0	0	552
Malaysia	19,884	70.38	36	0.18	28,251
Mongolia	0	0.00	0	0	2780
Philippines	125,975	134.00	654	0.52	94,013
Singapore	5330	102.82	6	0.11	5184
Viet Nam	69,680	81.00	61	0.09	86,025
Total for subregion	**241,112**	**13.46**	**837**	**0.35**	**1,791,271**
Pacific subregion					
Australia	820	3.67	0	0	22,342
Cook Islands	0	0.00	0	0	23
Fiji	245	28.69	0	0	854
French Polynesia	12	4.46	0	0	269
Marshall Islands	1257	2327.78	0	0	54
Federated States of Micronesia	1024	994.17	2	0.20	103
New Caledonia	1	0.41	0	0	246
New Zealand	42	1.01	0	0	4143
Palau	334	1590.48	0	0	21
Vanuatu	33	14.10	0	0	234
Total for subregion	**3768**	**13.32**	**2**	**0.05**	**28,289**
TOTAL	244,880	13.46	839	0.34	1,819,560

Source: WHO
† denotes the territories which are used in table

As per the Ministry of Health of Malaysia, it was surveyed that in 2012 DHF was only 3% in comparison with DF which was recorded with 97%. In another survey it was recorded that 77% of people from urban areas were suffering from dengue virus in comparison with rural area which was 23% only. Since it is an alarming situation to get over dengue virus, many researchers have contributed to understand dengue virus in terms of medical sciences; many research were conducted to understand the behavior of dengue virus. Some research were conducted to evaluate the spatial risk factor of dengue fever. Researchers also look into remote sensing and GIS parameters to create mapping for dengue incidences. Many statistical research also took place

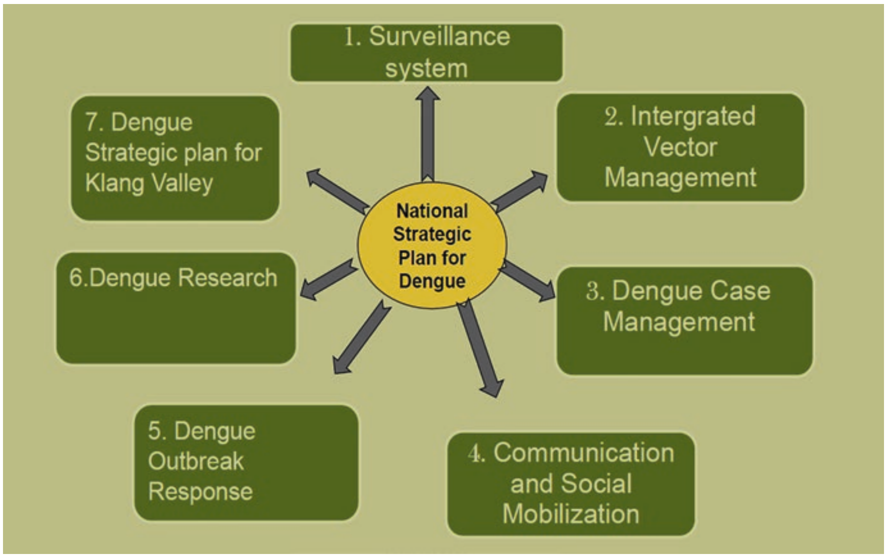

1. Surveillance system
7. Dengue Strategic plan for Klang Valley
2. Intergrated Vector Management
National Strategic Plan for Dengue
6.Dengue Research
3. Dengue Case Management
5. Dengue Outbreak Response
4. Communication and Social Mobilization

Fig. 4 Malaysian Government strategic plans for dengue [30]

and scored 70.3% significance in predicting dengue cases [35]. As per research, Malaysia have strong suggest dengue cases in area of forest based on serological results, data say 2.300 wild and domestic animals are representing over 55 species of more than 28 genera, over 25,000 adult mosquitoes are collected from urban forest [36].

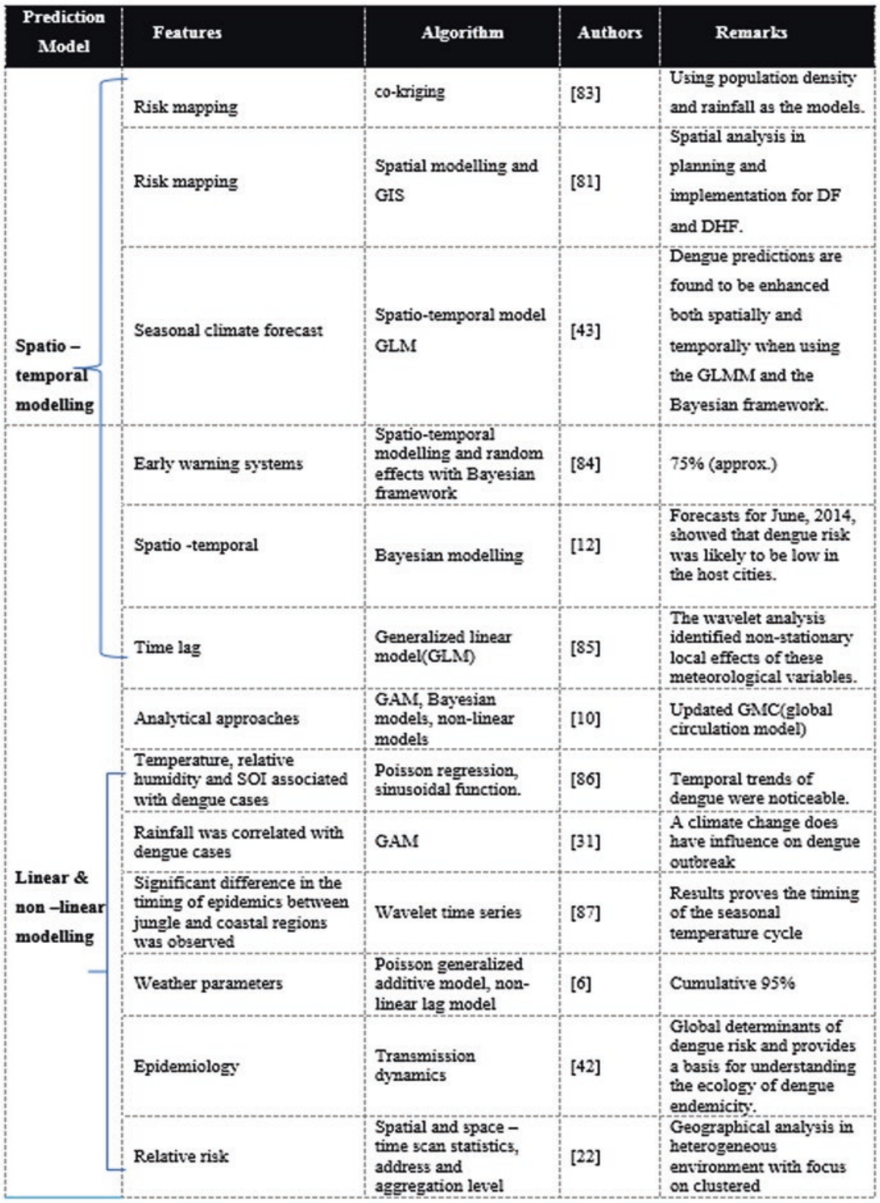
Table 2 shows the comparison table based on different techniques used for dengue prediction. Different research have focused on different concepts to understand the behavior of dengue virus. Since, from the applied methodes from existing research, Malaysia still do record many cases; hence, the proposed research is used to understand the aspect of dengue in Malaysia. A spatial mapping will be developed to predict and visualize the dengue incidences. Based on the map, information will be passed to the fogging team for them to focus on particular areas instead of spotting blindly.

3 Area of Study

This research is conducted for Malaysia, as vector control fails to control the endemic disease.

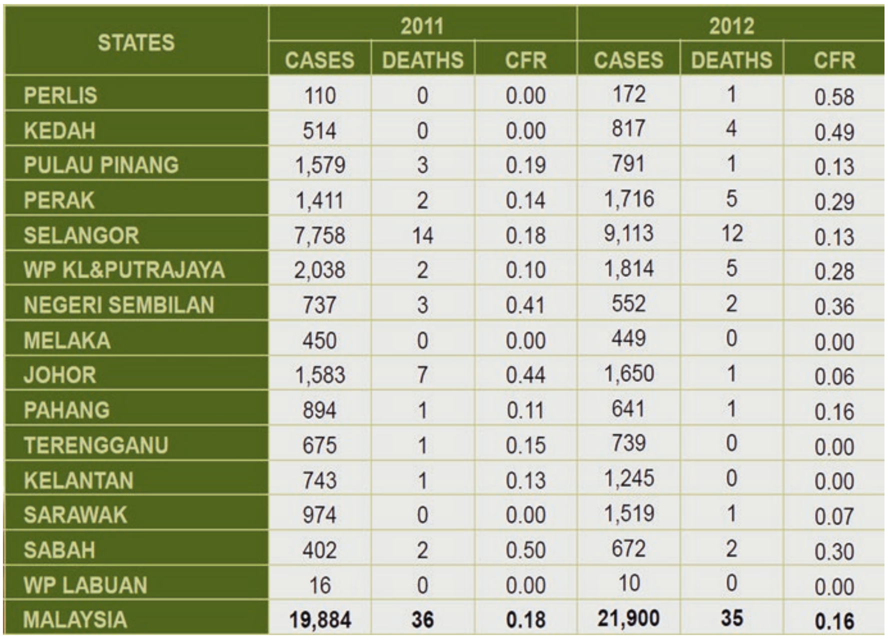
As shown in Fig. 5, Selangor had reported the maximum number of cases. 40% of the total cases of the whole country is recorded from Selangor from 2010 to 2012 [38].

Table 2 Dengue outbreak prediction methods and other techniques proposed by other researchers



Prediction Model	Features	Algorithm	Authors	Remarks
	Risk mapping	co-kriging	[83]	Using population density and rainfall as the models.
	Risk mapping	Spatial modelling and GIS	[81]	Spatial analysis in planning and implementation for DF and DHF.
Spatio – temporal modelling	Seasonal climate forecast	Spatio-temporal model GLM	[43]	Dengue predictions are found to be enhanced both spatially and temporally when using the GLMM and the Bayesian framework.
	Early warning systems	Spatio-temporal modelling and random effects with Bayesian framework	[84]	75% (approx.)
	Spatio -temporal	Bayesian modelling	[12]	Forecasts for June, 2014, showed that dengue risk was likely to be low in the host cities.
	Time lag	Generalized linear model(GLM)	[85]	The wavelet analysis identified non-stationary local effects of these meteorological variables.
	Analytical approaches	GAM, Bayesian models, non-linear models	[10]	Updated GMC(global circulation model)
	Temperature, relative humidity and SOI associated with dengue cases	Poisson regression, sinusoidal function.	[86]	Temporal trends of dengue were noticeable.
	Rainfall was correlated with dengue cases	GAM	[31]	A climate change does have influence on dengue outbreak
Linear & non –linear modelling	Significant difference in the timing of epidemics between jungle and coastal regions was observed	Wavelet time series	[87]	Results proves the timing of the seasonal temperature cycle
	Weather parameters	Poisson generalized additive model, non-linear lag model	[6]	Cumulative 95%
	Epidemiology	Transmission dynamics	[42]	Global determinants of dengue risk and provides a basis for understanding the ecology of dengue endemicity.
	Relative risk	Spatial and space – time scan statistics, address and aggregation level	[22]	Geographical analysis in heterogeneous environment with focus on clustered

As per the Statistical Department of Malaysia, Selangor is one of the major cities in Malaysia situated in Peninsular Malaysia. Malaysia has a total of 13 states. Selangor is toward the west side, surrounded by three other states, namely, Negeri Sembilan from the south, Perak from the north, and Pahang from the east, and an



STATES	2011			2012		
	CASES	DEATHS	CFR	CASES	DEATHS	CFR
PERLIS	110	0	0.00	172	1	0.58
KEDAH	514	0	0.00	817	4	0.49
PULAU PINANG	1,579	3	0.19	791	1	0.13
PERAK	1,411	2	0.14	1,716	5	0.29
SELANGOR	7,758	14	0.18	9,113	12	0.13
WP KL&PUTRAJAYA	2,038	2	0.10	1,814	5	0.28
NEGERI SEMBILAN	737	3	0.41	552	2	0.36
MELAKA	450	0	0.00	449	0	0.00
JOHOR	1,583	7	0.44	1,650	1	0.06
PAHANG	894	1	0.11	641	1	0.16
TERENGGANU	675	1	0.15	739	0	0.00
KELANTAN	743	1	0.13	1,245	0	0.00
SARAWAK	974	0	0.00	1,519	1	0.07
SABAH	402	2	0.50	672	2	0.30
WP LABUAN	16	0	0.00	10	0	0.00
MALAYSIA	**19,884**	**36**	**0.18**	**21,900**	**35**	**0.16**

Fig. 5 Malaysia dengue cases reported for 2011–2012 [40]

ocean named Straits of Malacca from the west. The demographic map can be visualized in Fig. 6. Selangor's total area is 8104 km^2 with population of 5,411,324 and population density of 670/km^2. Selangor is subdivided into nine districts, namely, Hulu Langat, Hulu Selangor, Gombak, Klang, Kuala Selangor, Kuala Langat, Sabak Bernam, Petaling Jaya, and Sepang.

For this research weather data was also collected from different auxiliary weather stations (AWS) which are situated around or in Selangor. From the weather data, it was estimated that Selangor has 80% relative humidity (RH) throughout the year, and mean rainfall was 2500 mm [41]. After looking deeper into the data, it was clear to say that Petaling Jaya had the maximum number of dengue cases with respect to other districts in Selangor. Whereas the total area of Petaling Jaya is comparatively small, i.e., 97.2 km2 and total population of 638,516. As per the Meteorological Department of Petaling Jaya, the overall temperature is warm (with some hot day) with sunny day and relatively cool in the evening. The average temperature can be seen between 23 °C and 33 °C.

As per the statistics, it is clear that from 2009 to 2013, an exponential growth of dengue cases has been reported. Ninety-three (about 90%) hotspots have been found in Petaling District of Selangor [37]. A survey was conducted on 10 July 2014, which stated 4007 cases reported for dengue in Kula Lumpur and Putrajaya with 7 deaths, and in compression with Selangor total cases were 16,441 and 36 were reported death.

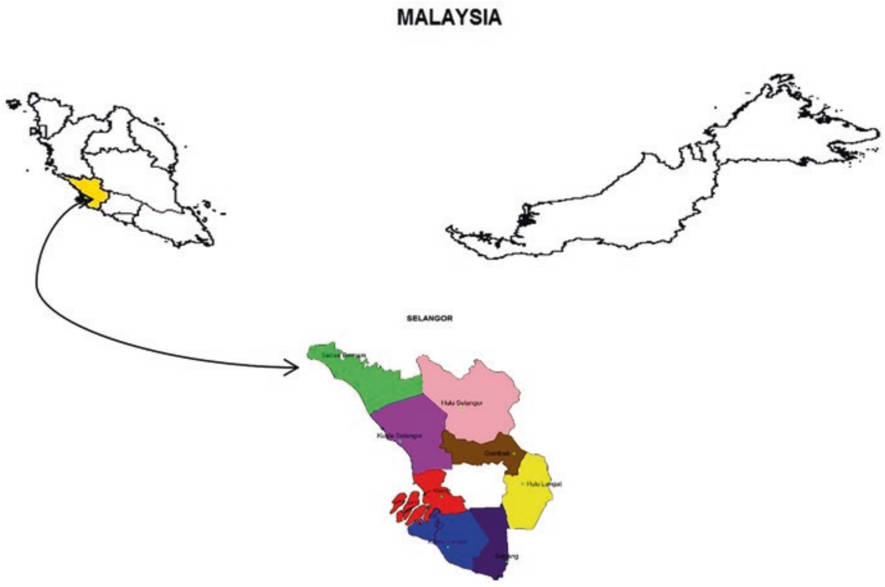


Fig. 6 Malaysia boundary map with all 13 states; highlighted is Selangor State and its subcities. (Source: Map is developed in R software using GIS tool and shape file (GADM))

4 Data Collection

Data plays a very important role in data prediction. Hence, a collection of data was performed for this research from many sources.

Firstly data was collected from the Ministry of Health (MoH) of Malaysia. Data include all the information about patients including notification ID, year (epid tahun), epid miggu, coordinates (latitude and longitudes), area (daerah), zone (mukim), and type of diagnosis performed.

Once data is collected from the sources, it is required to do data filtering based on the data structure. For example in our cases, received data is mixed with latitude and longitude since filling is required to get original location. Once the filtering is done, it is time to perform the predictive algorithms to get future prediction.

4.1 Mathematical Modelling Using Machine Learning

In the latest trend, machine learning has taken place for all processes and computations. Learning can be performed either supervised or unsupervised based on data set and requirement. This works well when it gets blended with mathematical modeling. The mathematical model is widely used in all fields of science, engineering, commerce, business, and many more places. Mathematical model helps to devlop and design the dynamic systems, of dynamics systems, statistical models or game theory, and many more. This research also focuses on probabilistic modelling to predict dengue cases by using unsupervised learning [42].

4.2 Gaussian Mixed Modelling

Mixing models are a type of density model that includes a number of component functions. Gaussian mixture model (GMM) is a parametric probability density function (PDF) that is represented as a weighted sum of Gaussian component densities. GMM parameters are estimated using training data estimated from a well-trained prior-model iterative expectancy maximization (EM) or maximum a posteriori (MAP) algorithm [43]. Gaussian mixture models can also be seen as a form of a generalized radial base function network in which each Gaussian component is a base function or a "hidden" unit.

Mathematically we define the GMM as

$$p(x \mid \Phi) = \sum_{m}^{i=1} \alpha_i p_i (x \mid \phi_i) \tag{1}$$

where, $x = (x_i, \ldots, x_d)^T \in R^d$, $\phi_i = (\mu_i, \Sigma_i)$, $\Phi = (\alpha_i, \ldots, \alpha_m, \phi_i, \ldots, \phi_m) \epsilon\, \Omega$ and each p_i is a d-dimensional multivariate Gaussian distribution given by

$$p_i (x \mid \phi_i) = \frac{1}{(2\pi)^{d/2} (\det \Sigma_i)^{1/2}} e^{-1/2(x-\mu_i)^T \sum_{-1}^{i}(x-\mu_i)} \tag{2}$$

where Σ_i is the d x d symmetric positive definite covariance matrix that corresponds the ith values. The collection of $\alpha_i^{'}s$ is known as the model's *mixture proportions*, i.e., here, all α_i represents the probability that a randomly selected. The probability density function (pdf) can be denoted as follows [44]:

$$\sum_{m}^{i=1} \alpha_i = 1$$

$$= \sum_{m}^{i=1} \alpha_i \int_{.}^{R^d} p_i (x \mid \phi_i) dx$$

$$= \int_{.}^{R^d} \sum_{m}^{i=1} \alpha_i p_i (x \mid \phi_i) dx$$

$$= \int_{.}^{R^d} p(x \mid \Phi) dx$$

To start the Gaussian mixture model, we need to define a likelihood estimate as follows: suppose us that we have an n-tuple of random $X = (X_1, X_2, \ldots, X_n)$ vectors that have been generated separately and equally distributed (iid). We assume also that the distribution of X depends on the unknown fixed parameters, i.e., $\theta = (\theta_{1,}\ldots, \theta_k)$ which take its values into the parameter space $\boldsymbol{\Theta}$. Since $X_i \in$ X the individual probability density function (pdf) for X_i is denoted as

$$X_i \sim f\left(x_i \mid \theta\right)$$

for $i = 1, \ldots, n$. So the joint pdf can be denoted as

$$f\left(x \mid \theta\right) = \prod_{n}^{i=1} f\left(x_i \mid \theta_{1,\ldots,}\theta_k\right) \tag{3}$$

Here, x = (x_1, …x_n) denotes the observation values of $X_1, \ldots, X_n$. Now we can compute maximum likelihood estimation (MLE) for the given Gaussian model. Let $f(x| \theta)$ denote the pdf of random samples X = (X_1, …X_n) as defined in Eq. 3.

The *likelihood function* is defined as

$$L\left(\theta \mid \mathrm{x}\right) = L\left(\theta_1, \ldots \theta_k \mid x_1, \ldots x_n\right) = f\left(\mathrm{x} \mid \theta\right) = \prod_{n}^{i=1} f\left(x_i \mid \theta_{1,\ldots,}\theta_k\right) \tag{4}$$

To understand the likelihood function, we must know the difference between $L(\theta|$ x) and f(x| θ). If we talk about f(x| θ), here θ is assumed as the unknown fixed quantities, whereas x is allowed to vary overall possible values in our sample space. Whereas in likelihood function $L(\theta|$ x), we treat x as known quantities, and θ is allowed to vary over the parameter space $\boldsymbol{\Theta}$. Now, after finding the likelihood of GMM, we compute the maximum likelihood, compute the maximum likelihood from observed samples x,

$$\hat{\Theta}\left(\mathrm{x}\right) \in {}^{argmax}_{\theta \epsilon \Theta} L\left(\theta \mid \mathrm{x}\right) \tag{5}$$

Here, $\hat{\Theta}\left(\mathrm{x}\right)$ is a maximum likelihood estimation for θ based on x, and *argmax* $L(\theta|$ x) denotes the set of all $\theta \epsilon\, \Theta$ which maximize $L(\theta|$ x) over the parameter space Θ. After looking at the basic Gaussian model, now we will move to another step where we will do maximum likelihood estimation and produce some cluster to locate the estimation, since clustering method is discussed in following part.

The clustering of data, also referred to as cluster analysis, segment analysis, taxonomy analysis, or non-supervised classification, is a way of creating groups of items or clusters in a way that allows for very specific objects and objects [45]. Since then, various data representative methods have created an extensive and sometimes confusing range of clustering methods, which have been able to measure proximity (similarity) between elements [46]. It is important to understand the difference between clustering and discriminant analysis. In clustering analysis mostly unsupervised classification technique is used, where the problem is to group a given collection of unlabeled patterns into meaningful cluster, and in discriminate analysis, we have supervised classifications which have collection of labeled patterns where problem is to label a newly encountered, yet unlabeled pattern. Clustering can be performed by several tasks such as gene expression data, clustering in health psychology and biomedical research, clustering in market research, and clustering in image segmentation. The data clustering problem has been addressed extensively, although there is no uniform definition for data clustering, and there may be on [47–49].

Data clustering process can be designed by following the clustering process, which has the following four stages: data representation, modeling, optimization, and validation [50]. This is shown in Fig. 7. In the first process, i.e., data representation predetermines what kind of cluster structure can be discovered in the data, since the modelling phase defines the notations of clusters and the criteria that separate the desired group structures from unfavorable ones [45]. After the best selection of data representation, the modeling process is done which proves what kind of clustering modeling can be used.

The clustering problem modelling can be divided into different clustering models, and these can be divided into other several submodels as shown in Fig. 8. After the selection of model, the next process comes for the optimization of model, after the optimization the validation of the followed model is validated. Clustering is used in mining data, in pattern recognition, and in other biological applications. A general approach to clustering is to view the density estimation problem. In density estimation-based clustering, pdf is estimated for given data set to search the regions that are densely populated. There are several algorithms to solve this problem; some

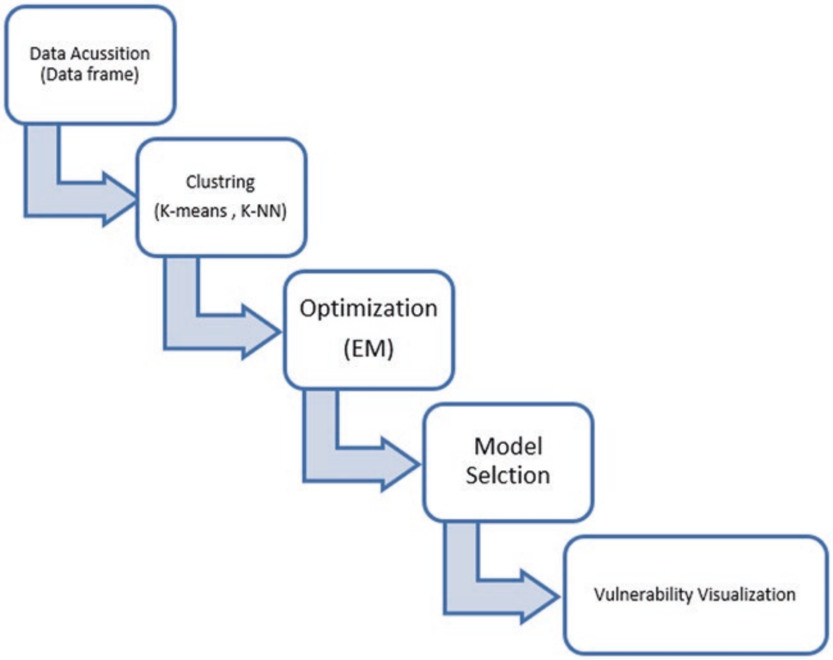


Fig. 7 Basic data cluster flow chart

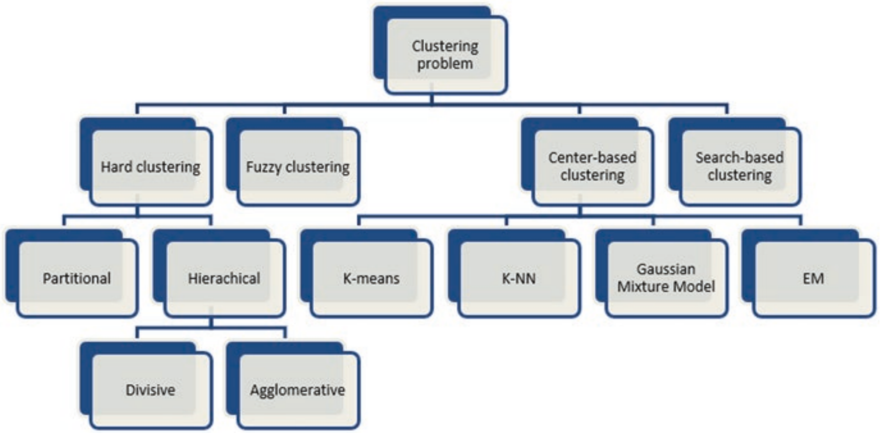


Fig. 8 Flow chart of clustering algorithm

widely used algorithms are EM algorithm and *k*-means clustering algorithm which are mentioned in following chapters [51].

Clustering problem can be solved in many different ways such as by hard clustering, fuzzy clustering, center-based clustering, search-based clustering, and many other clustering processes. In this research we have used center-based clustering, since compared with other type of clustering method, center-based clustering is very efficient for clustering large database and high-dimension database. Since, in this section *k*-means algorithm is described which is one of the most used clustering algorithm, which was first described by Macqueen [52].

4.3 The k-means Algorithm

The *k*-means algorithm is classified as a partitioned or nonhierarchical clustering method [53]. In this algorithm, the number of clusters k is assumed to be fixed, and there is an error function in this algorithm which proceeds for a given initial k clusters, by allocating the remaining data to the nearest cluster and then repeatedly changing the membership of the cluster according to the error function until the error function does not change significantly or the membership of the cluster no longer changes. The conventional *k*-means algorithm can be described as follows [54, 55]:

$$= \sum_{k}^{i=1} \sum_{x \in c_i} d\left(x, \mu\left(c_i\right)\right) \tag{6}$$

In Eq. 1, let D be the data set with n instances, and let c_1, c_2, c_3…. , c_k be the k disjoint cluster of D. Here, $\mu(c_i)$ is the centroid of cluster c_i, and $d(x, \mu(c_i))$ denotes the distance between x and $\mu(c_i)$.

The algorithm can be divided into two parts: the initialization phase and the iteration phase. At the time of initialization phase, the algorithm randomly assigns the cases into k cluster, and in iteration phase, the algorithm computes the distances between each case and each cluster and assigns the case to the nearest cluster. The objective function can be defined as shown in Eq. 2:

$$P(W,Q) = \sum_{k}^{I=1} \sum_{n}^{i=1} w_{il} d_{euc}(x_i, q_l) \tag{7}$$

where $Q = \{q_l, l = 1, 2\ldots, k\}$ is a set of objectives, $d_{euc}(., ..)$ is the Euclidean distances, and W is an n X k matrix that satisfies the following conditions:

1. $w_{il} \in \{0, 1\}$ for i = 1,2…,, l = 1,2…,k,
2. $\sum_{k}^{l=1} w_{il} = 1$ for i = 1, 2,…,n.

5 *k-means* Algorithm

Require: Data set D, Number of cluster k, Dimensions d:

1. Initialization Phase

```
1: $(C_1, C_2, …. C_k)$ = Initial partition of D.
2: (2. Iteration Phase)
3: **repeat**
4: $d_{ij}$ = *distance between case i and cluster j*;
5: $n_i = \arg min_{1 \le j \le k} d_{ij}$;
6: Assign case i to cluster $n_i$;
7: Recomputed the cluster means of any changed cluster above;
8: **until** no further changes of cluster membership occurs in a complete iteration
9: Output results.
```

5.1 k-means Clustering Algorithm to Create Initial Vulnerability Map

k-means clustering is an unsupervised learning algorithm well known for clustering problem. It focuses on the K centroid of each cluster; all centroids are placed at different locations. Then each point belonging to data is associated to the nearest centroid; if no point is pending, then group age is done. Then we recalculate K new centroid as barycentre of each clusters. Finally the loops will be created, and we can notice the centroids are moving step by step until no more changes are done.

$$J = \sum_{k}^{j=1} \sum_{x}^{i=1} \| x_i^{(j)} - c_j \|^2 \tag{8}$$

where $\| x_i^{(j)} - c_j \|^2$ is distance measure between data point $x_i^{(j)}$ and cluster centre c_j.

To implement the abovementioned formula in our data set, the first step we took is as follows: Randomly initialize the K cluster centers in the input data set. After that we randomly pick up the data point π_i from the input data set, and for j = 1,2,3…k, calculate the class membership function I(x_i,j). Every point is assigned to the cluster whose centroid is the closest to that point. After that for all K cluster centers, set c_i to be the center of mass of all points in cluster C_i.

$c_i = \frac{1}{c_i} \sum_{x_i \in c_i} x_i$ Repeat this process until all cluster centers remain unchanged or until they change to some threshold value. The stopping threshold value is usually selected as being very small, or set an upper number of iterations to certain threshold values, since the center of the cluster will be known.

5.2 *K-Nearest Neighbors' Algorithm (K-NN)*

K-NN is a machine learning algorithm that is based on the Euclidian distance between test samples and the specified training samples. K-NN is used as clasification technique for the data set which is splited in to testing set and training set, vector are found and the classification is done via the maximum of summed kernel densities.

In K-NN algorithm, the neighbors are the closest samples in the feature space. The number of neighbor samples is defined as k which depends on the characteristics of data. Larger values of k reduce the effect of noise but also make boundaries between classes less distinct. The output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.

In this study, k = 3, 5, 8, and 10 were chosen using Euclidian distance for the classification based on the performance of the previous study.

5.3 *Expectation Maximization (EM) Algorithm*

EM algorithm is a broadly applicable approach to the iterative computation of maximum likelihood (ML) estimations, useful in a variety of incomplete data problems [56]. Data that have missing values are known as *censored* data. These data might obtain some incomplete information but have useful other information. The EM algorithm has the ability to deal with missing data and unidentified variables, so it is becoming useful in a variety of incomplete-data problems. The EM algorithm can be applied not only to incomplete -data set, but aslo work perfectly where data is missing, truncated distribution, or censored or grouped observation, but also a whole variety of situations where the incompleteness of the data is not all that natural or evident. On each iteration of the EM algorithm, there are two steps: *Expectation* step or *E*-step and the *Maximization* step or *M* -step.

The basic idea of EM algorithm is to associate with the given incomplete-data problems, a complete-data problems for which ML estimation is computationally more tractable. Now we formulate the general EM algorithm; let say we have random incomplete data set X which have some probability distribution and Y which is unobserved observations, so the final data set can be defined by $Z = (X, Y)$ which is a complete data set. Now, to drive this further, we make some notations as follows: let $f(z| \Phi) = f(x, y| \Phi)$ represent the joint pdf of random variable X and Y; we also took marginal pdf of X which is denoted by $g(\mathrm{x}| \Phi)$. We also let conditional probability distribution of Y which is denoted as $k(y| x, \Phi)$. The main aim of EM is to maximize the incomplete data log likelihood, which is denoted as

$$\log\left[L\left(\Phi \mid X\right)\right] = \log\left[g\left(x \mid \Phi\right)\right] \tag{9}$$

over $\Phi \in \Omega$, by using relationship between $f(x, y| \Phi)$ and $g(\mathrm{x}| \Phi)$. Now, basic equation can be developed as

$$f\left(z \mid \Phi\right) = f\left(x, \mid y, \mid \Phi\right) = k\left(y \mid, x \mid, \Phi\right) g\left(\mathrm{x} \mid \Phi\right) x \in X, y \in Y \tag{10}$$

Equation number 9 can be used for EM algortihm for classification process, by finding the first step called E- step which refer to find the expected values of complete data log likelihood, which can be defined as $log[L(\Phi| X)] = \ log\ [g(x| \Phi)]$ and relating some current parameters for estimation $\Phi^c \in \Omega$.

Thus,

$$Q\left(\Phi \mid \Phi^c\right) = \mathbb{E}\left[logf\left(x, \mid y, \mid \Phi\right) \mid, x \mid, \Phi^c\right] x \in X, y \in Y, \Phi^c \in \Omega \tag{11}$$

where Q represents the expectation of log likelihood and $E\left[.\right]$ denotes expectation operator. Now moving on to the second step which is M-step which seeks the maximization of E-step, i.e., we chose $\Phi^+ \in \Omega$ such that,

$$\Phi^+ \in \underset{\theta \epsilon \Theta}{argmax} Q\left(\Phi \mid \Phi^c\right) \tag{12}$$

The general EM iteration is given as follows:

(1) E-step: Calculate $Q(\Phi| \Phi^{c.})$.
(2) M-step: Choose $\Phi^+ \in \underset{\theta \epsilon \Theta}{argmax} Q\left(\Phi \mid \Phi^c\right)$.
(3) Let $\Phi^c = \Phi^+$.
(4) Repeat (1)–(3) as necessary.

5.4 Model Selection for EM Algorithm

To use the EM algorithm, it is necessary to select the perfect model which can best fit to available data set. There are many model criteria for selection of the best model. In this research we have used Bayesian information criterion (BIC) as the model selection among the finite set of models. The selection is done on the basis of the calculated values of BIC; if the model has the lowest BIC value, it is considered to be the best fitting model.

Formally BIC can be defined by the following equation:

$$BIC = -2\ln\hat{L} + k\left(\ln\left(n\right)\right) \tag{13}$$

and

$$\hat{L} = p\left(x \mid \hat{\theta}.M\right) \tag{14}$$

where,

$\hat{\theta}$ = parameter values that maximize the likelihood function
x= the observed data
θ= the parameters of the model
n= the number of data points in x
k= the number of free parameters to be estimated
$\hat{L}$ = the maximized value of the likelihood function

5.5 Results and Discussion

This section will visualize the results generated from all the abovementioned methods. The data was break into weekly data, and clusters were assumed initially by assigning process say (k-3, 5, 8, 10).

Table 3 shows the values computed for *k*-means for different k values. In this table data set is divided into 3-month category. The first is January, July, and December.

5.6 K-Nearest Neighbor (K-NN) Classification Results

To validate the *k*-means values, K-NN classifier is used to estimate the best fit of the algorithm. In Table 4 all different data set have been used to calculate the K-NN values based on different k numbers. This table also shows the mean squared error values and best kernel shape used to fit data.

Table 3 *k*-means result based on sum of square for different k values with the months of January, July, and December

Data set	Algorithm	K = 3	K = 5	K = 8	K = 10
January (2014 Week 1 N = 219	Sum of square %	63.1%	84.2%	89.1%	91.4%
January (2014) Week 2 N = 428	Sum of square %	61.1%	77.7%	86.4%	88.3%
January (2014) Week 3 N = 595	Sum of square %	54.7%	69.3%	87.7%	89.7%
January (2014) Week 4 N = 636	Sum of square %	61.0%	81.3%	85.9%	88.5%
July (2014) Week 1N = 509	Sum of square %	61.9%	73.0%	86.0%	88.5%
July (2014) Week 2 N = 661	Sum of square %	59.4%	81.6%	87.7%	90.8%
July (2014) Week 3 N = 526	Sum of square %	66.9%	77.7%	90.0%	92.7%
December (2014) Week 1 N = 520	Sum of square %	64.8%	80.7%	87.9%	90.8%
December (2014) Week 2 N = 663	Sum of square %	66.4%	82.0%	88.1%	91.3%
December (2014) Week 3 N = 868	Sum of square %	64.8%	81.6%	88.8%	90.8%
December (2014) Week 4 N = 1085	Sum of square %	64.3%	80.7%	86.9%	89.8%

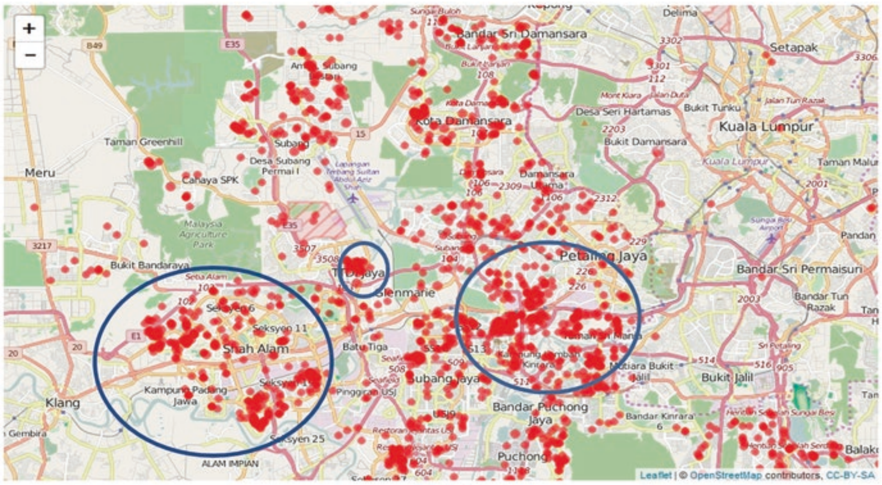
Table 4 K-NN values calculated with best kernel fit

Data set	Algorithm	Best kernel	Mean squared error	Number of k
January (2014) Week 1 N = 219	K-NN	Rectangular	0.002864456	10
January (2014) Week 2 N = 428	K-NN	Triangular	0.002699823	10
January (2014) Week 3 N = 595	K-NN	Optimal	0.002977822	10
January (2014) Week 4 N = 636	K-NN	Rectangular	0.002257728	9
July (2014) Week 1 N = 509	K-NN	Optimal	0.00245263	10
July (2014) Week 2 N = 661	K-NN	Optimal	0.002359995	8
July (2014) Week 3 N = 526	K-NN	Triangular	0001981571	10
December (2014) Week 1 N = 520	K-NN	Triangular	0.002353327	10
December (2014) Week 2 N = 663	K-NN	Triangular	0.0023470.37	10
December (2014) Week 3 N = 868	K-NN	Triangular	0.002521785	10
December (2014) Week 4 N = 1085	K-NN	Triangular	0.002572776	10

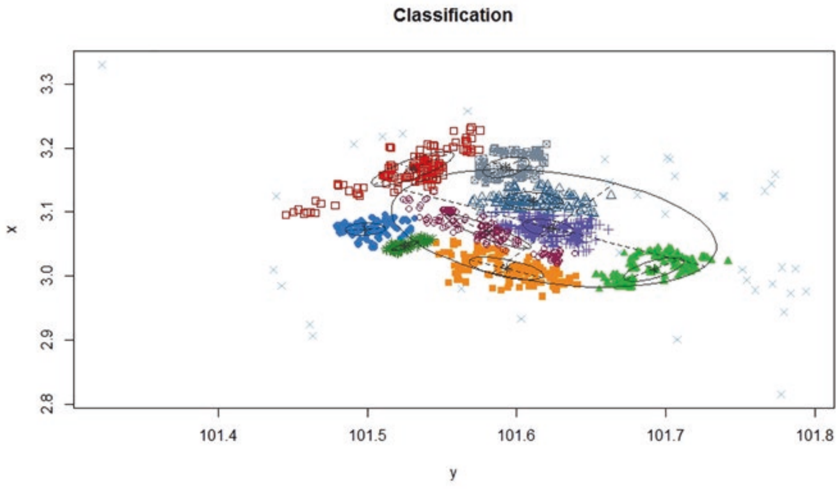
Estimation maximization algorithm was computed by using R software in which Mclust package was used to generate results. As shown in Figs. 9, 10, and 11 all visualize the dynamic mapping based on EM clusters. Based on the cluster density and cluster radius, some can be marked as hotspot for future prediction. The coordinates 101.5 to 101.7 and 3.1 to 3.29 are highly dense and clustered.

5.7 Density Plot

As shown in Figs. 12, 13, 14, 15 and 16, all plots are used to visualize the density function based on coordinates and dengue incidences. In the density plot for the January 2014 data set, it is noted that some clusters have high peak point which can



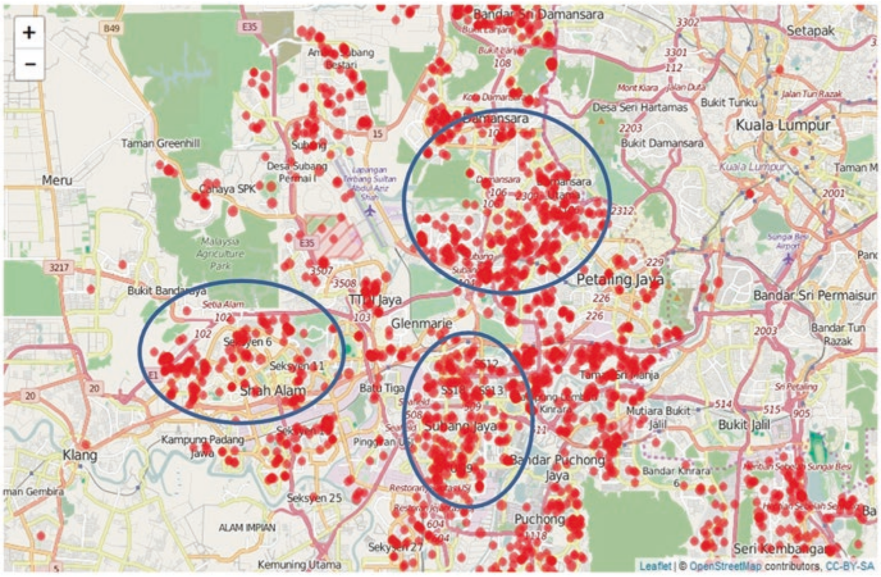
(a)



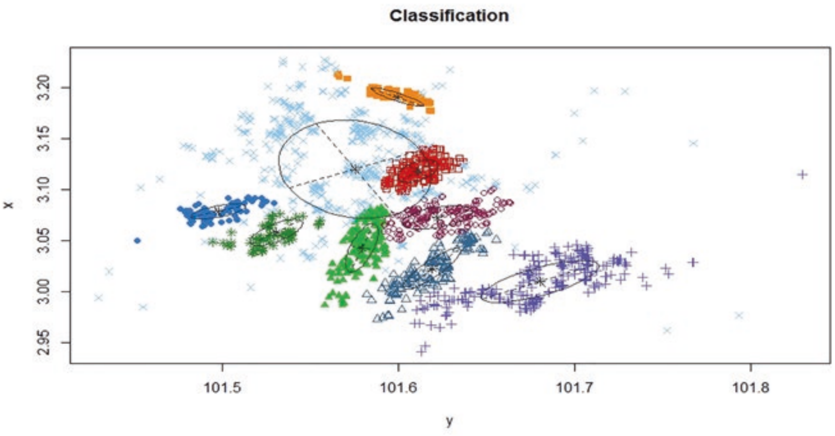
(b)

Fig. 9 (**a**) shows that the dengue incidences occurred in January 2014 and the circled areas are highly infected areas; as from figure (**b**) dengue incidences are clustered using EM, which represents areas from 101.55 3.10 3.11 to 101.655 3.20 are highly infected

be mapped as hotspot point to correlate the hotspot; the cluster plot is mentioned just above the density plot to locate the correct location. From the data set, first week of January 2014 has some dense cluster, among them the highest peak density is noted from 101.45 to 101.55 and from 3.00 to 3.15, and for the second week, from 101.6 to 101.7 and 3.0 to 3.2 are the most dense areas. Moving further with the third



(a)

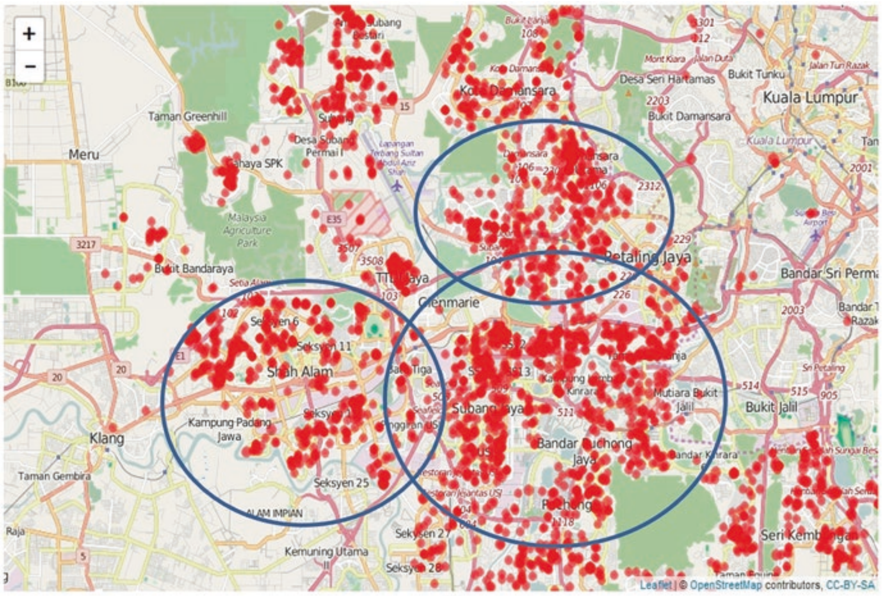


(b)

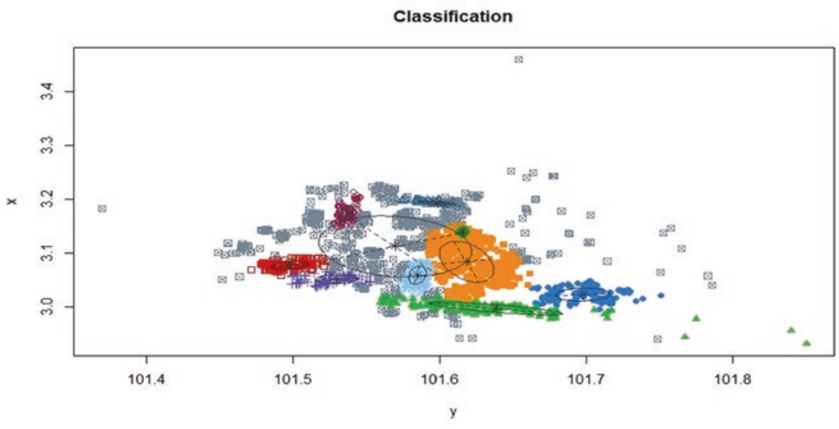
Fig. 10 (**a**) shows that dengue incidences occurred in July 2014 and the circled areas are highly infected areas; as from figure (**b**) dengue incidences are clustered using EM, which represents areas from 101.59 3.10 3.00 to 101.700 3.15 are highly infected

week notice that the coordinates were from 101.45 to 101.60 and 3.00 to 3.20 and for the fourth week 101.50 to 101.65 and 3.05 to 3.10.

From Fig. 17 we can visualize the time series movement and correlate it with human activities, in Fig. 17a it shows the dengue incidences lies in the month of January 2014 since it has the big number of reported dengue incidences which is



(a)



(b)

Fig. 11 (**a**) shows that dengue incidences occurred in December 2014 and the circled areas are highly infected areas; as from figure (**b**) dengue incidences are clustered using EM, which represents areas from 101.58 2.90 to 101.65 3.30 are highly infected

followed by the monsoon season also have Chines new year as shown in figure (d) is in same month since we can visualizes the movement of incidences from one point to other, in figure (b) shows the June 2014 incidences which were also highly reported incidences, since at time Hari raya as shown in figure (e) also celebrated since the lot of movement can be notice from one point to another due to public

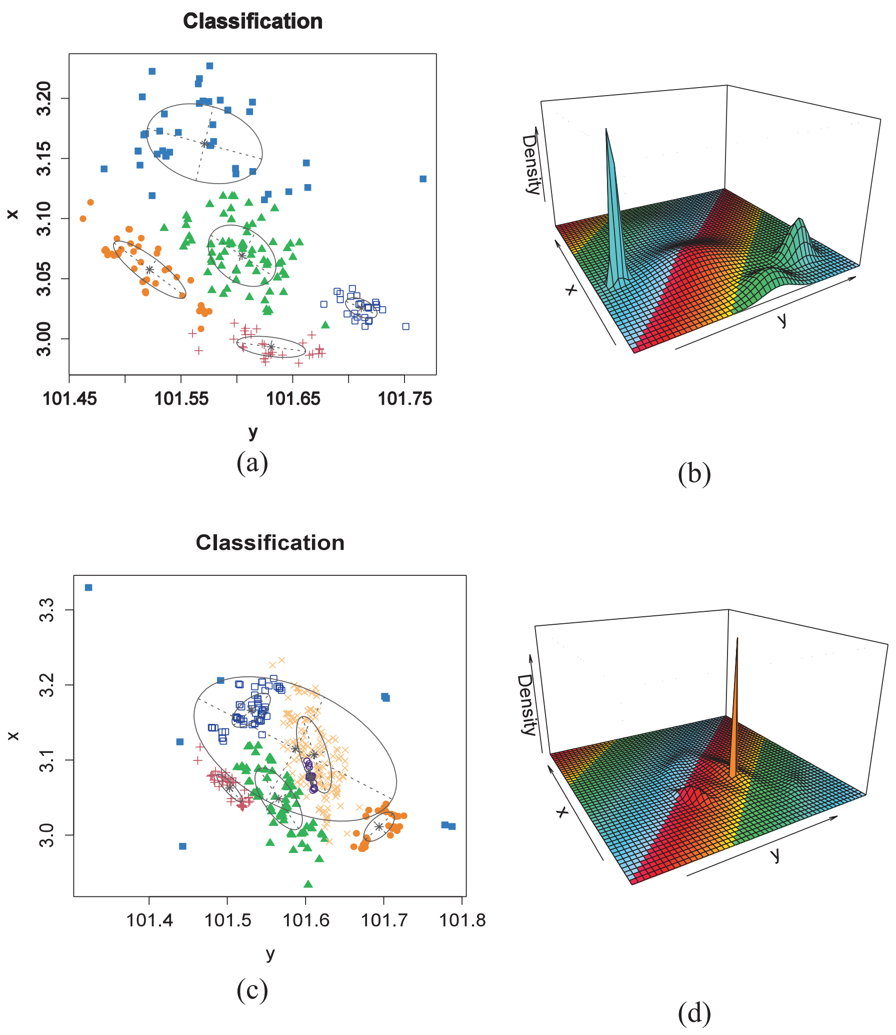


Fig. 12 (**a**) shows the cluster based on EM for January 2014 first week; the density of incidences is shown in (**b**) which shows the high dense area is 3.05 101.50 (Kampung Padang Jawa); (**c**) shows the cluster plot for the second week of January 2014 in correlation with density in (**d**) for area 3.1 101.59 (Kelang Jaya)

holidays (c) shows the dengue incidences occurred in December 2014, which also have many public holidays. It is visualized that places from coordinates 2.95 101.3 to 3.69 101.65 have the maximum number of dengue incidences; few colleges and schools lie in this area, since the movement of incidences can be noticed from mentioned areas.

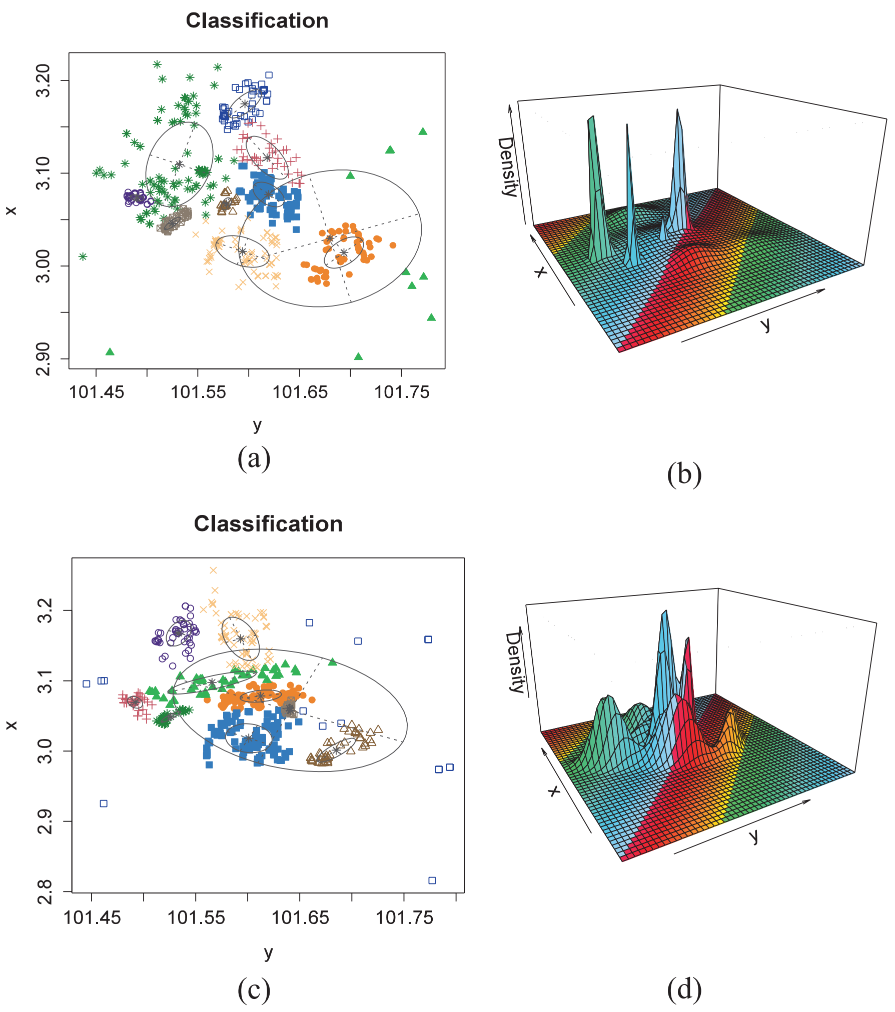
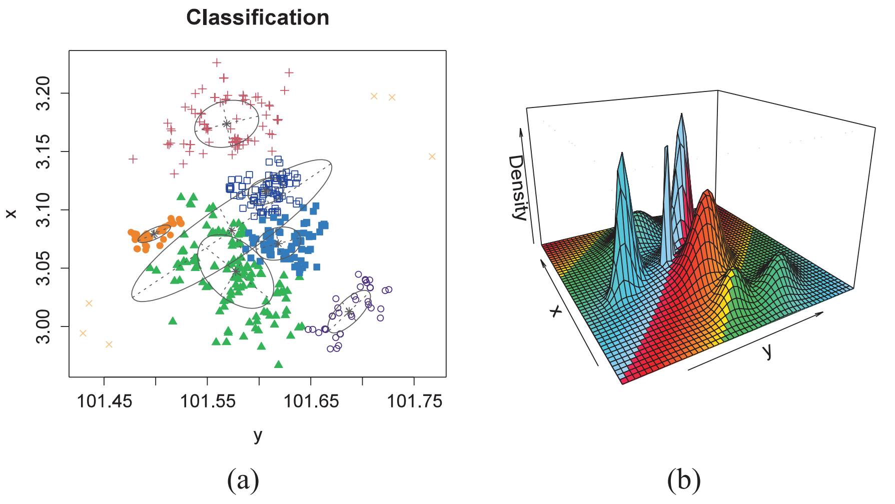


Fig. 13 (**a**) shows the cluster based on EM for January 2014 third week; the density of incidences are shown in (**c**) which shows the high dense area is 3.00 101.51 (Taman Sepakat), 3.10 101.55(Ilham apartment), and 3.05 101.58 (Persian Perpaduan); (**b**) shows the cluster plot for the fourth week of January 2014 in correlation with density in (**d**) for area 3.09 101.59 to 3.0 101.49 (U1 shah alam)



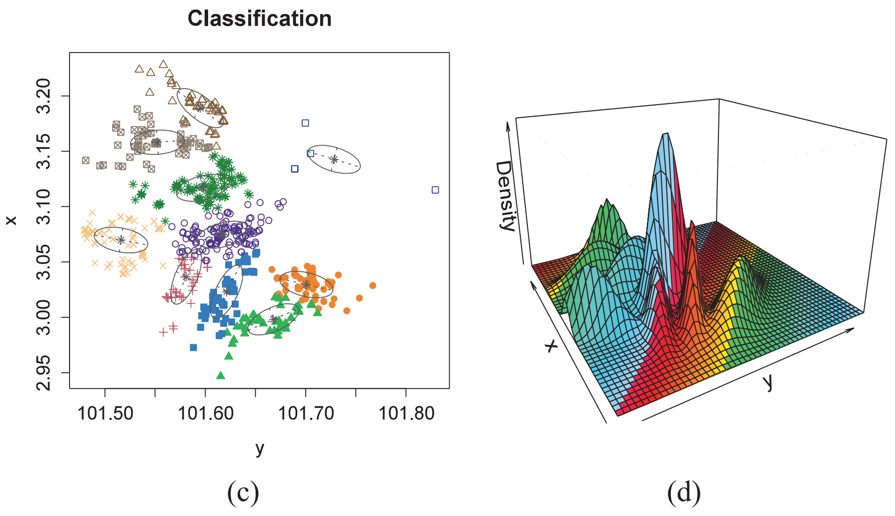
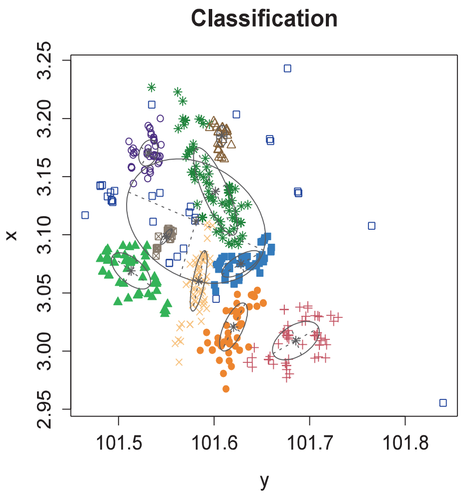
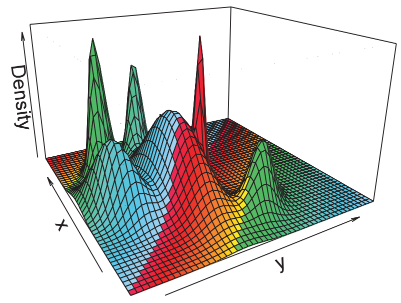


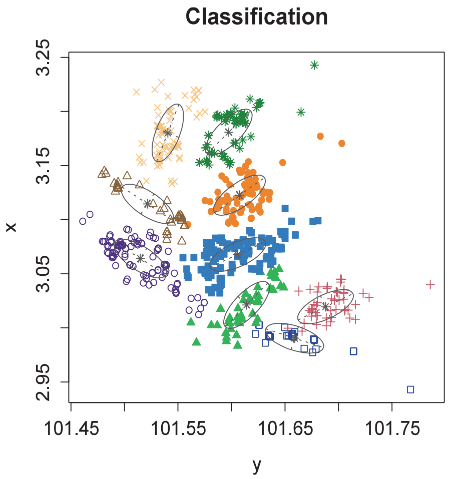
Fig. 14 (**a**) shows the cluster based on EM for July 2014 first week; the density of incidences are shown in (**b**) which shows the high dense area is 3.09 101.63 (seksyen 51); (**c**) shows the cluster plot for the second week of July 2014 in correlation with density in (**d**) for area 3.06 101.53 to 3.07 101.63 (Seksyen 15)



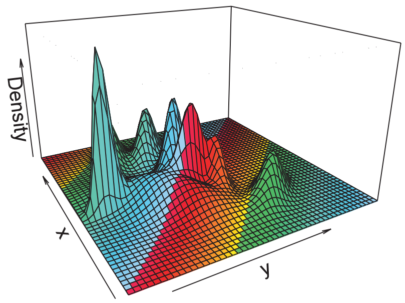
(a)



(b)



(c)



(d)

Fig. 15 (**a**) shows the cluster based on EM for December 2014 first week; the density of incidences are shown in (**b**) which shows the high dense area is 3.06 101.52 to 3.15 101.63 (seksyen 2); (**c**) shows the cluster plot for the second week of December 2014 in correlation with density in (**d**) for area 2.98 101.67 to 3.13 101.70 (Taman lestari permai, Taman kota perdana, Taman pinggiran putra)

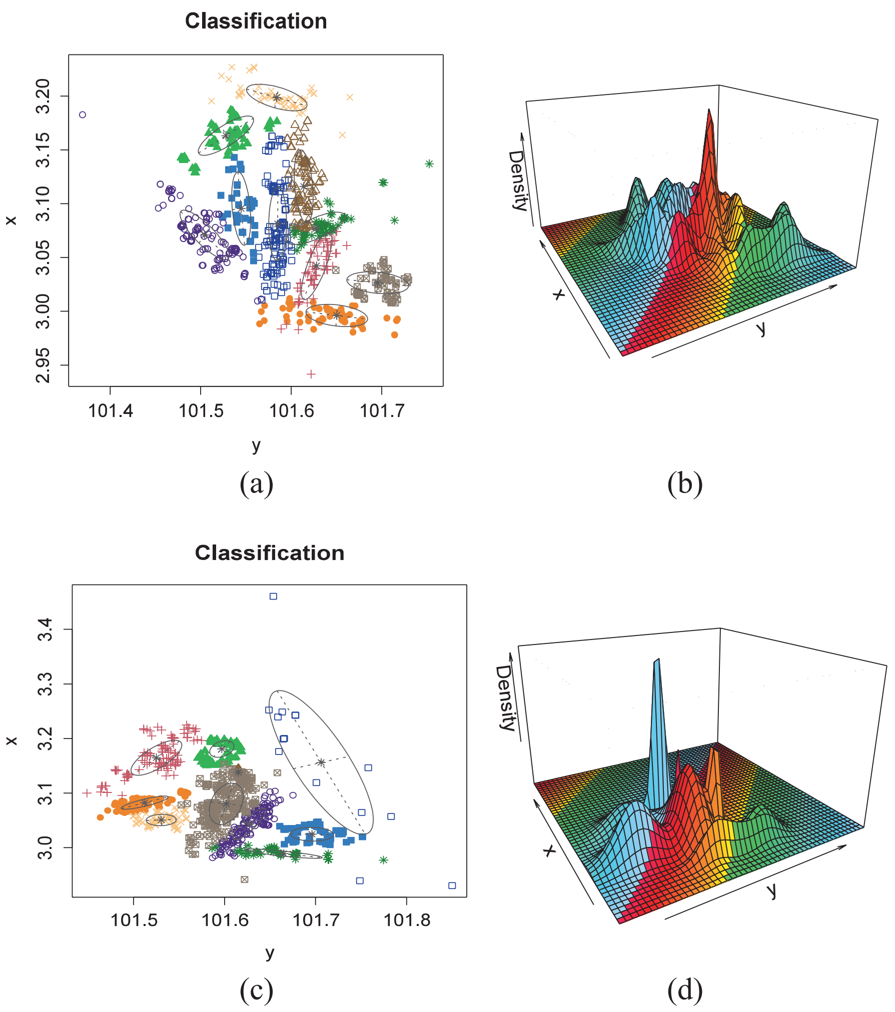


Fig. 16 (**a**) shows the cluster based on EM for December 2014 third week; the density of incidences are shown in (**b**) which shows the high dense area is 3.07 101.58 to 3.06 101.61 (Bandar sunway;) (**c**) shows the cluster plot for the fourth week of December 2014 in correlation with density in (**d**) for area 3.00 101.59 to 3.11 101.63 (Taman Paramount, Wilayah Persekutuan, KL)

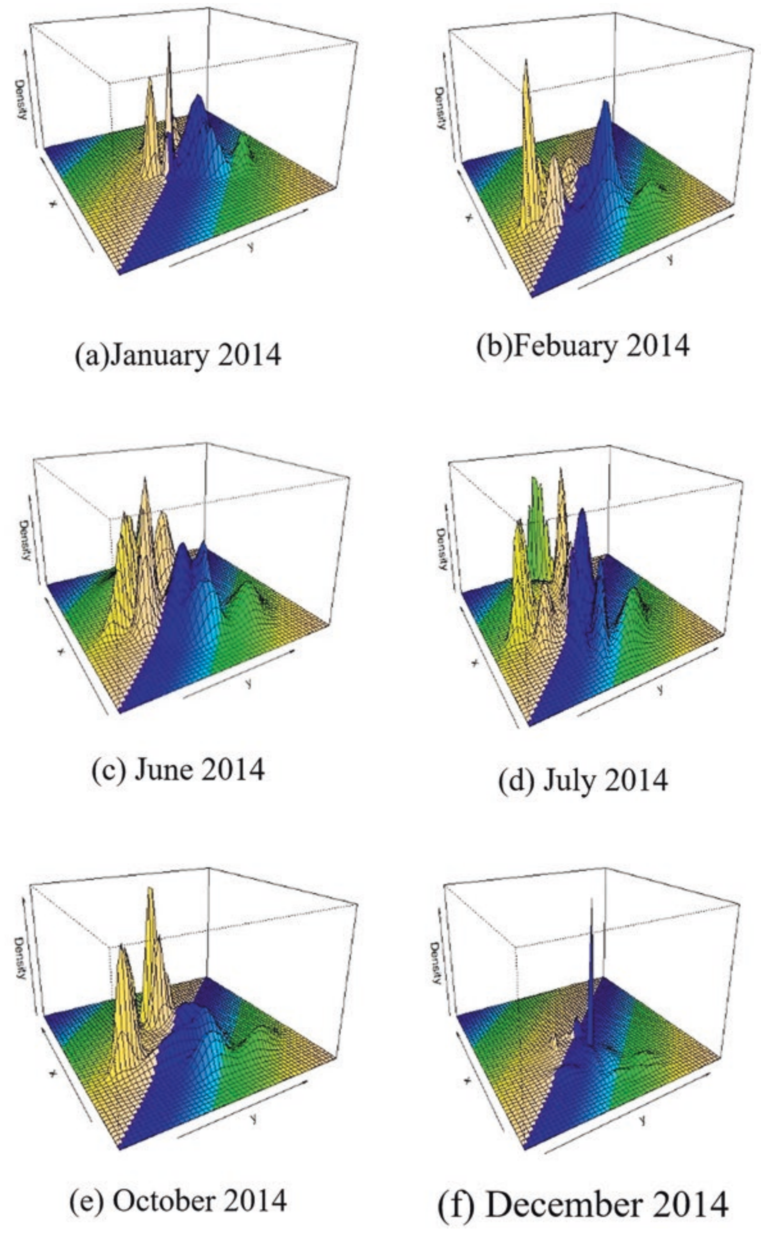


Fig. 17 (**a**) shows the dengue incidences for January 2014; (**b**) shows the dengue incidences that occurred in the month of February 2014; (**c**) dengue incidences occurred in June 2014; (**d**) incidences occurred in July 2014; (**e**) number of incidences reported in October 2014; (**f**) dengue incidences in December 2014

6 Conclusion and Discussion

This study proposes the visualization of dengue reported cases in Malaysia in the province of Selangor State and its districts. In 2014, the highest dengue outbreaks were recorded as the rise of incidences was incited from the month of May till June; the maximum number of incidences can be localized for the particular coordinates. These results support the finding for the spatial mapping for Petaling District and its sub-districts named as Bukit Raja, Sungai Buloh, Damansara, and Petaling, as per the provided and collected data from MoH, Malaysia clarifies the maximum number of dengue incidences for particular time period. On the basis of the mapping and the clustering algorithms, we can see a baseline on which we can justify the dengue occurrence. Hence, the identified clusters at sub-district and district of Petaling may be driven by the human mobility rather than the spatial action on vector part [57, 58]. After one week of rainfall, the maximum number of dengue cases are notified. To control the dengue incidences, we have focused on the prediction mapping techniques to improve the risk management for dengue outbreak [59].

The surveillance and data with accuracy is the back bone for any mapping model, in this research we focused on the state Selangor, and perfromed sureillance for this area and collected data for the area. The suggested model for the predictions provides strong evidence to state and map the dengue incidences. The number of cluster for the mentioned area was high, and other depending variables such as log likelihood and other mentioned algorithms were computed for better results [60]. The coordinate range from 3.00 to 3.15 latitude and longitude from 101.45 to 101.70 were found in most of the clustered area. Hince, the visualization of the vulnerability mapping can help to generate early warning for dengue incidences with the help of mapping techniques using *k*-means for predicting k values to validate the centroid *k*-means we proposed the classifier function which is very widely used named as K-NN. After analyzing the results of *k*-means, we extend to EM algorithm from where we were able to locate the classes of each cluster, and for the best selection to fit the model, we used BIC model selection. Hence, the EM algorithm was able to locate the correct location for dengue incidences as these coordinates will be proposed to vector control unit for the fogging and to control the dengue virus [61].

References

1. Gubler, D.J.: Dengue and dengue hemorrhagic fever. Clin. Microbiol. Rev. **11**(3), 480–496 (1998)
2. Halstead, S.B.: Dengue. Lancet. **370**, 1644–1652 (2007)
3. Bhatt, S., Gething, P.W., Brady, O.J., Messina, J.P., Farlow, A.W., Moyes, C.L., et al.: The global distribution and burden of dengue. Nature. **496**, 504–507 (2013)
4. Siler, J.F., Hall, M.W., Kitchens, A.: Dengue: its history, epidemiology, mechanism of transmission, etiology, clinical manifestations, immunity and prevention. Philippine Bur. Sci. (1926)
5. Cheong, Y.L., Burkart, K., Leitão, P.J., Lakes, T.: Assessing weather effects on dengue disease in Malaysia. Int. J. Environ. Res. Public Health. **10**, 6319–6334 (2013)

6. Chen, C., Nazni, W., Lee, H., Seleena, B., Mohd Masri, S., Chiang, Y., et al.: Mixed breeding of Aedes aegypti (L.) and Aedes albopictus Skuse in four dengue endemic areas in Kuala Lumpur and Selangor, Malaysia. Trop. Biomed. **23**, 224–227 (2006)
7. Karim, M.M.: Dengue Fever: a tropical disease coming to Europe. Well-founded concern and the need for concerted action. BioMed Cent. (2014)
8. Chen, S.-C., Liao, C.-M., Chio, C.-P., Chou, H.-H., You, S.-H., Cheng, Y.-H.: Lagged temperature effect with mosquito transmission potential explains dengue variability in southern Taiwan: insights from a statistical analysis. Sci. Total Environ. **408**, 4069–4075 (2010)
9. Naish, S., Dale, P., Mackenzie, J.S., McBride, J., Mengersen, K., Tong, S.: Climate change and dengue: a critical and systematic review of quantitative modelling approaches. BMC Infect. Dis. **14**, 167 (2014)
10. Mathur, N., Asirvadam, V.S., Dass, S.C.: Spatial-temporal visualization of dengue incidences using Gaussian Kernel. In: 2018 International Conference on Intelligent and Advanced System (ICIAS), pp. 1–6. IEEE (2018)
11. Khormi, H.M., Kumar, L.: Assessing the risk for dengue fever based on socioeconomic and environmental variables in a geographical information system environment. Geospat. Health. **6**, 171–176 (2012)
12. Lowe, R., Barcellos, C., Coelho, C.A., Bailey, T.C., Coelho, G.E., Graham, R., et al.: Dengue outlook for the World Cup in Brazil: an early warning model framework driven by real-time seasonal climate forecasts. Lancet Infect. Dis. **14**, 619–626 (2014)
13. Rudnick, A., Lim, T.W., Ireland, J., Perubatan, I.P.: Dengue fever studies in Malaysia. Institut Penyelidikan Perubatan, Kuala Lumpur (1986)
14. Moncayo, A.C., Fernandez, Z., Ortiz, D., Diallo, M., Sall, A., Hartman, S., et al.: Dengue emergence and adaptation to peridomestic mosquitoes. Emerg. Infect. Dis. **10**, 1790–1796 (2004)
15. Bi, P., Tong, S., Donald, K., Parton, K.A., Hobbs, J.: Climate variability and the dengue outbreak in Townsville, Queensland, 1992-93. Environ. Health. **1**, 54 (2001)
16. Skae, F.M.T.: Dengue fever in Penang. Br. Med. J. **2**, 1581–1582 (1902)
17. Rudnick, A., Tan, E.E., Lucas, J.K., Omar, M.: Mosquitoborne haemorrhagic fever in Malaya. Br. Med. J. **1**, 1269–1272 (1965)
18. Smith, C.E.G.: Isolation of three strains of type 1 dengue virus from a local outbreak of the disease in Malaysia. J. Hyg. **54**, 569–580 (1956)
19. Smith, C.E.G.: A localized outbreak of dengue fever in Kuala Lumpur: epidemiological and clinical aspects. Med. J. Malaya. **10**, 289–303 (1956)
20. Smith, C.E.G.: A localized outbreak of dengue fever in Kuala Lumpur: serological aspects. J. Hyg. **55**, 207–223 (1957)
21. Gagnon, A.S., Bush, A.B., Smoyer-Tomic, K.E.: Dengue epidemics and the El Niño southern oscillation. Clim. Res. **19**, 35–43 (2001)
22. García, C., García, L., Espinosa-Carreón, L., Ley, C.: Abundancia y distribución de Aedes aegypti (Diptera: Culicidae) y dispersión del dengue en Guasave Sinaloa, México. Int. J. Trop. Biol. Conserv. **59** (2010)
23. Gomes, A.F., Nobre, A.A., Cruz, O.G.: Temporal analysis of the relationship between dengue and meteorological variables in the city of Rio de Janeiro, Brazil, 2001–2009. Cad. Saude Publica. **28**, 2189–2197 (2012)
24. Rozilawati, H., Zairi, J., Adanan, C.: Seasonal abundance of Aedes albopictus in selected urban and suburban areas in Penang, Malaysia. Trop. Biomed. **24**, 83–94 (2007)
25. Li, C., Lim, T., Han, L., Fang, R.: Rainfall, abundance of Aedes aegypti and dengue infection in Selangor, Malaysia. Southeast Asian J. Trop. Med. Public Health. **16**, 560–568 (1985)
26. Fairos, W.W., Azaki, W.W., Alias, L.M., Wah, Y.B.: Modelling Dengue Fever (DF) and Dengue Haemorrhagic Fever (DHF) outbreak using poisson and negative binomial model. Int. J. Math. Comput. Sci. Eng. **4**, 809–814 (2010)
27. W. H. Organization, S. P. f. Research, T. i. T. Diseases, W. H. O. D. o. C. o. N. T. Diseases, W. H. O. Epidemic, P. Alert: Dengue: Guidelines for Diagnosis, Treatment, Prevention and Control. World Health Organization (2009)

28. M. M. S. Malaysia.: Department of Statistics, Malaysia. (2014).
29. Azami, N.A.M., Salleh, S.A., Neoh, H.-m., Zakaria, S.Z.S., Jamal, R.: Dengue epidemic in Malaysia: not a predominantly urban disease anymore. BMC. Res. Notes. **4**, 216 (2011)
30. M. o. H. M. N. d. s. p. 2009–2013: Ministry of Health Malaysia (2009). National dengue strategic plan 2009–2013. 2009.
31. M. o. H. M. A. R. 2012: Ministry of Health Malaysia. Annual Report 2012. (2012).
32. Khormi, H.M., Kumar, L.: Modeling dengue fever risk based on socioeconomic parameters, nationality and age groups: GIS and remote sensing based case study. Sci. Total Environ. **409**, 4713–4719 (2011)
33. Lee, H., Chen, C., Masri, S.M., Chiang, Y., Chooi, K., Benjamin, S.: Impact of larviciding with a Bacillus thuringiensis israelensis formulation, Vectobac WG®, on dengue mosquito vectors in a dengue endemic site in Selangor state, Malaysia. Southeast Asian J. Trop. Med. Public Health. **39**, 601–609 (2008)
34. Elder, J., Lloyd, L.: Achieving Behaviour Change for Dengue Control: Methods, Scaling-Up, and Sustainability. World Health Organization, Geneva (2006)
35. Shafie, A.: Evaluation of the spatial risk factors for high incidence of dengue fever and dengue hemorrhagic fever using GIS application. Sains Malaysiana. **40**, 937–943 (2011)
36. Rudnick A.: Studies of the ecology of dengue in Malaysia: a preliminary report l, 2. (1965).
37. Chowell, G., Cazelles, B., Broutin, H., Munayco, C.V.: The influence of geographic and climate factors on the timing of dengue epidemics in Perú, 1994-2008. BMC Infect. Dis. **11**, 164 (2011)
38. Feldstein, L.R., Brownstein, J.S., Brady, O.J., Hay, S.I., Johansson, M.A.: Dengue on islands: a Bayesian approach to understanding the global ecology of dengue viruses. Trans. R. Soc. Trop. Med. Hyg., trv012 (2015)
39. Ling, C.Y., Gruebner, O., Krämer, A., Lakes, T.: Spatio-temporal patterns of dengue in Malaysia: combining address and sub-district level. Geospat. Health. **9**, 131–140 (2014)
40. M. o. H. M. N. d. s. p. 2009–2013: Ministry of Health Malaysia (2009). National dengue strategic plan 2009–2013. (2009).
41. Olaniyi, A.O., Abdullah, A.M., Ramli, M.F., Sood, A.M.: Factors affecting agricultural land use for vegetables production—A case study of the state of Selangor, Malaysia. Afr. J. Agr. Res. **7**, 5939–5948 (2012)
42. Cummings, D.A., Iamsirithaworn, S., Lessler, J.T., McDermott, A., Prasanthong, R., Nisalak, A., et al.: The impact of the demographic transition on dengue in Thailand: insights from a statistical analysis and mathematical modeling. PLoS Med. **6**, e1000139 (2009)
43. Reynolds, D.: Gaussian mixture models. In: Encyclopedia of Biometrics, pp. 659–663. Springer (2009)
44. Plasse, J.H.: The EM Algorithm in Multivariate Gaussian Mixture Models using Anderson Acceleration. Worcester Polytechnic Institute (2013)
45. Gan, C.M., Wu, J.: Data Clustering: Theory, Algorithms, and Applications, vol. 20. Siam (2007)
46. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. (CSUR). **31**, 264–323 (1999)
47. Estivill-Castro, V.: Why so many clustering algorithms: a position paper. ACM SIGKDD Explor. Newsl. **4**, 65–75 (2002)
48. Dubes, R.C.: How many clusters are best?-an experiment. Pattern Recogn. **20**, 645–663 (1987)
49. Fraley, C., Raftery, A.E.: How many clusters? Which clustering method? Answers via model-based cluster analysis. Comput. J. **41**, 578–588 (1998)
50. Arbib, M.A.: The Handbook of Brain Theory and Neural Networks. MIT Press, Cambridge, MA (2003)
51. Ranganathan S.: Improvements to k-means clustering. (2013).
52. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, **1**(4), 281–297 (1967)

53. Jain, A.K., Dubes, R.C.: Algorithms for clustering data. Prentice-Hall, Inc, Upper Saddle River (1988)
54. Hartigan, A.: Clustering Algorithms. Wiley, New York (1975)
55. Hartigan, J.A., Wong, M.A.: Algorithm AS 136: A k-means clustering algorithm. Appl. Stat., 100–108 (1979)
56. McLachlan, G., Krishnan, T.: The EM Algorithm and Extensions. Wiley, Hoboken (382, 2007)
57. Teurlai, M., Huy, R., Cazelles, B., Duboz, R., Baehr, C., Vong, S.: Can human movements explain heterogeneous propagation of dengue fever in Cambodia? PLoS Negl Trop Dis. **12** (2012)
58. Lin, C.-C., Huang, Y.-H., Shu, P.-Y., Wu, H.-S., Lin, Y.-S., Yeh, T.-M., et al.: Characteristic of dengue disease in Taiwan: 2002–2007. Am. J. Trop. Med. Hyg. **82**, 731–739 (2010)
59. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
60. Kanagachidambaresan, G.R., Mahima, V., Prakash, K.B.: Programming tensor flow with single board computers, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_12
61. Vadla, P.K., Ruwali, A., Lakshmi, M.V.P., Kanagachidambaresan, G.R., Neural network, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_5

Vector-Borne Disease Outbreak Prediction Using Machine Learning Techniques

Sandali Raizada, Shuchi Mala, and Achyut Shankar

1 Introduction

The World Health Organisation recognises that more than 17% of all infectious diseases is accounted for by vector-borne diseases. Around the world, in 97 countries malaria transmission occurs which places about 3.4 billion people at risk. The world's population of over 40% is at risk of dengue. In 2006, the outbreak spread of chikungunya was in several countries, including India, where 1,400,000 cases were reported. Countries like the United States of America reported 51,258 vector-borne disease cases in 2013. The Global Vector Control Response recognises vector control as the elementary approach to prevent vector-borne diseases and as a response in order to control outbreak. The Indian Government announced US$ 9.87 billion outlay for the health sector and aimed to increase healthcare funds to 3% of the gross domestic product (GDP) by 2022. It is the need of the hour to approximate the outbreak of a vector-borne disease.

With evolution in data mining, [1] was early to use data mining techniques and neural network approach to analyse heart disease dataset. Shraddha Shivhare et al. [2] explored if the nucleus alone is sufficient for classification of white blood cells using neural networks with fivefold cross-validation. Md. Osman Goni Nayeem et al. [3] proposed a neural network that has been used to first classify a patient as infectious or not using neural network, and later by using two hidden layers, the type of disease was predicted. The diseases considered were liver disorder, heart disease and lung cancer. A work implemented a prediction system to predict cardiac disease [4]. Marios Anthimopoulos et al. [5] put forward and evaluated a CNN

S. Raizada · S. Mala · A. Shankar (✉)
Department of Computer Science and Engineering, ASET, Amity University Uttar Pradesh, Noida, Uttar Pradesh, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_9

(convolutional neural network) which could be used for lung disease pattern classification, and another CNN for the diagnosis of chest diseases was proposed [6].

A dengue fever surveillance system was created by Agus Qomaruddin Munir et al. [7] which could be used for forecasting dengue fever spread in a particular area. His system was designed to contain the spread of dengue fever, and it was achieved by using three algorithms which helped in predicting the spread of dengue in a year. Juan M. Scavuzzo et al. [8] considered three diseases which had mosquito as its vector. The study depended on remote sensing and included temporal modelling of the oviposition activity based on time series of data extracted using satellite images. In addition to linear models, other machine learning techniques were also used to provide better results. B. Mahalakshmi et al. [9] focused on the prediction of Zika virus. The problem statement was to find the existence of Zika virus in a person with priority given to pregnant women. The data used was created through collecting the symptoms of the virus over the Internet, stored in the cloud, and then an artificial neural network (ANN) was used as a classifier. The purpose was to construct network with the ideal nodes which would provide a good-quality solution. An analysis with comparison of various other classification algorithms that purposed to work for the prediction of other diseases in the future was also performed.

V. Janani et al. [10] studied dengue prediction using data mining techniques and machine learning. A sequential minimal optimisation framework was proposed that suggested a network classification technique where the first latent affiliations of actors were captured by extracting disease prediction, and then extant data mining techniques were applied for classification. For symptom dataset of dengue disease, Weka tool was also used, and tests of different algorithms were done. Thus, it has been established that an ANN is one of the most powerful classifiers for tasks of medical diagnosis. However, annually, malaria kills over 1.2 million people, and the fastest-growing disease due to a vector is dengue fever.

Developments in machine learning technology has caused an increased amount of awareness onto using networks to either predict the type of disease in a person or classify them as an infectious or noninfectious. Most works focused on the history of the patient with no attention to climatic conditions of regions and living habits that go around in a region. The spread of disease can be correlated with vector population which depends upon biotic and abiotic environmental factors and can be used to predict an outbreak. A model suggested to predict for female mosquito population based on environmental data and also applied various validation methods [11]. Another found a relation between environmental factors and outbreak of the disease and supported a model that used classification algorithms. It was found that the second model support vector machine (SVM) was accurate and could be used for malaria outbreak prediction [12]. However, the minimal sets of characteristics that preserve a class were automatically selected. To the finest of our understanding, none of the existing works have worked on state-wise Indian subcontinent data spread over a period of 5 years by ANN. Furthermore, a wide contrasting range of features govern the outbreak.

In an attempt to restrict the spread of vector-borne diseases and to solve these problems, we explore contrasting characteristics comprising of environmental and social factors. First, for structured data, we seek advice and suggestions from experts to extract meaningful characteristics. Second, we predict the severity of outbreak in various regions of the country. Third, we use statistical analysis to determine the major vector-borne diseases. Finally, we propose an ANN multimodal disease outbreak prediction (ANN-MDOP) algorithm.

The rest of this study is arranged as follows: The data and description of model are outlined in Sect. 2. Then, the introduction of methods is provided in Sect. 3. The performance evaluation of ANN-MDOP algorithm is done in Sect. 4. The overall results are traced in Sect. 5. Section 6 is the final section which contains the conclusion of the work.

2 Dataset and Description of Model

The data used in this work are described. Furthermore, disease outbreak prediction model and methods for evaluation are also provided in this section.

2.1 Demographical Data

The demographic data used in this work is comprised of the positive cases of three diseases: chikungunya, malaria and dengue. The data is stored in the data centre. The data included the number of positive cases in the country arranged state-wise for every year. We used 5-year dataset from 2013 to 2017. Our data focused on 5,415,958 positive cases.

2.2 Meteorological Data

The meteorological data used contains real-life weather data which is stored in the data centre. The data provided by the weather department include temperature, humidity and rainfall over a span of 5 years from 2013 to 2017. Our focus is on state-wise dataset which includes 28 states and 8 union territories across the Indian subcontinent. The data contains two types of data, structured numerical data and some textual data.

Table 1 contains data collected from India. The data is classified and separated into two categories, namely, demographical data and meteorological data. With an objective to find out the prominent disease which affects the region of India, we have compiled a statistical analysis on the number of affected patients, average humidity, average temperature, average precipitation and the cases of major disease

Table 1 Item taxonomy in India demographical and meteorological data

Data category	Item	Description
Demographical data	Positive cases	Number of positive cases in a state
Meteorological data	Temperature	Annual temperature of a state
	Humidity	Annual humidity of a state
	Rainfall	Annual precipitation of a state

Table 2 Initial statistics data obtained in India

Statistics	2013	2014	2015	2016	2017
Number of positive cases	958,958	1,144,041	1,271,865	1,198,999	842,095
Average rainfall	65028.2	56416.3	59816.3	58912.1	66181.4
Average temperature	864.375	872.9028	884.5	887.2083	886.7639
Average humidity	2217.917	2118.74	2134.167	2136.083	2124.875

every year from the data. From Table 2, we can acquire the number of positive cases in a year and infer that the maximum number of patients was reported in 2015. Moreover, the meteorological data obtained by the weather department suggest that the weather of a region cannot be the only characteristic which determines vector population. In this work, we mainly focus on both abiotic and biotic features to predict the outbreak.

2.3 Disease Outbreak Prediction

The outbreak of the main vector-borne disease in the region can be acquired From Table 2. The purpose of this analysis is to envisage which areas of the region are in high risk, which are in moderate risk and which can be called in low-risk region. This model presents an excellent case of machine learning which uses supervised learning technique. The value provided as input is the feature value consisting of both biotic and abiotic factors, $X = (x_1, x_2, \ldots, x_n)$. This includes positive cases of three major vector diseases that prevail in the region, annual rainfall of the region and temperature and humidity of the region.

The visualisation or classification of an area in the low-, moderate- or high-risk region is dependent on the output value Y. $Y = \{Y_0, Y_1, Y_2\}$ where Y_0 indicates that the region is at low risk, Y_1 indicates that the region is at moderate risk and Y_2 indicates that the region is at high risk. We will next outline the spread of the dataset, characteristics of the dataset, setting of the experiment and learning algorithms briefly.

For dataset, with thorough conversation with experts, we focused on the following dataset to reach a closure:

- Positive and weather data (P&W-data): the positive case and weather data were multi-dimensionally fused to predict whether a region is at low risk, moderate risk or high risk.

In this experiment, we selected all the 28 states and 8 union territories of the Indian subcontinent and divided them into 60% training, 20% test set and 20% cross-validation. Hence, the ratio of train, cross-validation and test set is 6:2:2. Next, we use Python with TensorFlow to recognise the deep learning and machine learning techniques. In this work, for P-data, according to the statistics obtained, we extracted around 5,415,958 positive cases. Then, we divided these cases into cases of chikungunya, malaria and dengue to obtain three characteristics. For W-data, we first extracted weather data of 52 major cities in India and averaged it out to obtain the weather conditions of the state.

We next trace machine learning and deep learning techniques used in this work. We use three machine learning algorithms used widely for neural networks, i.e. feed-forward (FF), backpropagation (BP) and gradient descent (GD) algorithms, to predict the outbreak.

The network has a cost function and works by following the basic feed-forward, backpropagation and gradient descent ML algorithms to calculate and update the value of weights. The cost function and the algorithms are discussed below.

2.3.1 Cost Function

Every algorithm owns a cost function to analyse its performance. It is differentiable function used to quantify the error between predicted values and expected values. It is either minimised (returns error) or maximised (returns reward).

The cost function for neural network based on logistic cost function is given in (1)

$$\begin{aligned} J(\rho) = \frac{1}{m} \sum_{m}^{t=0} \sum_{K}^{k=0} [y_k^{(t)} \log\left(h_\rho\left(x^{(t)}\right)\right)_k \\ + \left(1 - y_k^{(t)}\right) \log\left(1 - h_\rho\left(x^{(t)}\right)_k\right) + \frac{\mu}{2m} \sum_{L-1}^{l=1} \sum_{a_l}^{i=1} \sum_{a_{l+1}}^{j=1} \left(\rho_{j,i}^{(l)}\right)^2 \end{aligned} \tag{1}$$

where L = total layers, a_l = unit number except the bias unit, K = number of output classes and $h\rho(x)k$ = hypothesis that results in the k^{th} output.

We add nested $\sum$ to hold them accountable for output nodes that are multiple. Part one of the equation talks about a nested $\sum$ that loops through the amount of output nodes. In the next part which is regularisation, we hold them accountable for multiple rho matrices. We then square every term.

2.3.2 Feed-Forward

Feed-forward algorithm works with the aim of calculating the value of the output neurons. It works by calculating the value of each neuron, at each layer (expect the input layer). The initial weights assigned to the network help the algorithm to determine the values

$$n_1^{(2)} = g\left(\rho_{10}^{(1)}x_0 + \rho_{11}^{(1)}x_1 + \rho_{12}^{(1)}x_2 + \rho_{13}^{(1)}x_3 \ldots\right) \tag{2}$$

$$n_2^{(2)} = g\left(\rho_{20}^{(1)}x_0 + \rho_{21}^{(1)}x_1 + \rho_{22}^{(1)}x_2 + \rho_{23}^{(1)}x_3 \ldots\right) \tag{3}$$

$$n_3^{(2)} = g\left(\rho_{30}^{(1)}x_0 + \rho_{31}^{(1)}x_1 + \rho_{32}^{(1)}x_2 + \rho_{33}^{(1)}x_3 \ldots\right) \tag{4}$$

$$h_\theta\left(x\right) = n_1^{(3)} = g\left(\rho_{10}^{(2)}n_0^{(2)} + \rho_{11}^{(2)}n_1^{(2)} + \rho_{12}^{(2)}n_2^{(2)} + \rho_{13}^{(2)}n_3^{(2)}\right) \tag{5}$$

where n = activation node, ρ = weight matrix and $x_0, x_1, \ldots, x_n$ = inputs.

The activation nodes for layer 2 with three neurons and n outputs from previous layer can be determined using (2), (3) and (4), and the final output containing one neuron taking output from layer 2 could be calculated using (5).

2.3.3 Backpropagation

The backpropagation algorithm works with an aim to minimise the error between the actual and predicted values. In order to achieve this, it calculates the error at each activation unit and updates the values of the weights. It thus minimises the cost function by calculating its partial derivative.

The steps are as under:

Given training set $\{(x_1, y_1)\ldots.\ (x_m, y_m)\}$
Set $\Delta_{i,j}^{(l)} = 0$ for all (l,i,j), create an all-zero matrix

For example in training t = 1 to m:

1. Start with setting $n^{(1)} = x^{(t)}$
2. Next, use feed-forward to compute $n^{(l)}$ for l starting from 2 to L

$$\text{Using } y^{(t)}, \text{ compute } \delta^{(L)} = n^{(L)} - y^{(t)} \tag{6}$$

where L is our total layers and $n^{(L)}$ is the vector of outputs of the active perceptron units for the final output layer.

$$\text{Compute } \delta^{(L-1)}\delta^{(L-2)}\ldots\ldots\delta^{(2)} \textit{using}\, \delta^{(l)} = \left(\left(\rho^{(l)}\right)^T \delta^{(l+1)}\right).*n^{(l)}.*\left(1-n^{(l)}\right) \tag{7}$$

A function called g-prime is then multiplied using element-wise multiplication. The function is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$:

$$g'\left(z^{(1)}\right) = n^{(1)} .* \left(1 - n^{(1)}\right) \tag{8}$$

$$\Delta_{i.j}^{(1)} = \Delta_{i.j}^{(1)} + n_j^{(1)} \delta_i^{(1+1)} \tag{9}$$

$$\text{Or with vectorisation, } \Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \left(n^{(l)}\right)^T \tag{10}$$

Hence we update our new Δ matrix:

$$D_{i.j}^{(l)} = \frac{1}{m}\left(\Delta_{i.j}^{(l)} + \mu \rho_{i.j}^{(l)}\right), \text{if } j \neq 0 \tag{11}$$

$$D_{i.j}^{(1)} = \frac{1}{m}\left(\Delta_{i.j}^{(1)}\right) \text{if } j = 0 \tag{12}$$

2.3.4 Gradient Descent

The algorithms let an assurance that the backpropagation is working as intended through gradient checking. Once we recognise that the backpropagation algorithm works fine, we need to turn it off as it may slow down our work.

With multiple weight matrices, we may represent our derivative of cost function with respect to θ_j:

$$\frac{\partial}{\partial \rho_j} J(\rho) \tilde{\sim} \frac{J\left(\rho_1, \ldots.., \rho_j + \epsilon, \ldots \rho_n\right) - J\left(\rho_1 \ldots.., \rho_1 - \epsilon, \rho_n\right)}{2\epsilon} \tag{13}$$

The pseudo codes for training a network using the above algorithms are:

1. Initialise the weights randomly.
2. Feed-forward propagation to $h_\rho(x^{(i)})$ for any $x^{(i).}$
3. Introduce cost function.
4. To compute partial derivatives, use backpropagation.
5. Next use gradient checking that checks if your backpropagation works. Then disable it.
6. Finally, use gradient descent to minimise the cost function with the weights in ρ.

Ultimately for P&W-data, the prediction of the outbreak and classification of the region by use of ANN-MDOP algorithm are done. In the next section, the particulars about ANN-MDOP are given.

2.4 Methods for Evaluation

For the purpose of evaluating performance, we denote true positive, true negative, false positive and false negative. Then, we obtain the measurement of accuracy, recall, precision and F1 scores

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+TN+FN}, \text{Recall} = \frac{TP}{TP+FN}, \text{Precision} = \frac{TP}{TP+FP}, \text{F1 score} = \frac{2 * Precision * Recall}{Precision + Recall}$$

where accuracy is the fraction of right predictions and precision and recall attempt to answer the correct proportion of positive identifications and identify the correct proportion of actual positives, respectively. The F1-score is calculated as the weighted harmonic mean of the recall and precision. For a multiclass classification, the average micro and average macro for the above values could be calculated.

We may regularise bias and variance and use learning curves to diagnose and gain better insights in addition to evaluation criteria.

Further we can also calculate the loss penalty for bad prediction. Formally, it indicates how bad the prediction was on an example. For classification we calculate the cross-entropy loss as follows:

$$\text{CE} = -\sum_{C}^{i} t_i \log(s_i)$$

For multiclass classification, this loss is softmax activation plus a cross-entropy loss as follows:

$$\text{Softmax} = \text{f}(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \tag{14}$$

$$\text{CE} = -\sum_{C}^{i} t_i \log\left(f(s)_i\right) \tag{15}$$

Get more training examples, try on smaller set of features and increase learning rate to fix high variance. Add features and polynomial features, and decrease learning rate to fix high bias. We focus on achieving low variance and low bias. Moreover, a neural network may be prone to underfitting or overfitting. A model with fewer parameters is prone to underfitting and is computationally cheaper. An algorithm with more parameters is prone to overfitting and is computationally expensive. Regularisation can be used in this case.

3 Methods

We have outlined the data imputation and normalisation and ANN multimodal outbreak prediction (ANN-MDOP) algorithm in this section.

3.1 Data Imputation and Normalisation

The positive case data contains some missing data values. Thus, we need to fill the data. In the very first step, we need to identify uncertain data and then alter them to improve the quality of the data. Then, we integrate data and execute pre-processing of data. For data imputation, we identify the null values and replace them with the mean of respective attributes. We, thus, fill in missing data and prepare the data for the model.

3.2 ANN-Based Multimodal Disease Outbreak Prediction (ANN-MDOP) Algorithm

For the purpose of data processing, we make use of ANN multimodal disease outbreak prediction (ANN-MDOP) algorithm. This can be broken down into six steps.

3.2.1 Text Data Representation

For each text word in the data, we convert it to a categorical variable, i.e. the text is represented in categories where each entry belongs to one or more categories (Fig. 1).

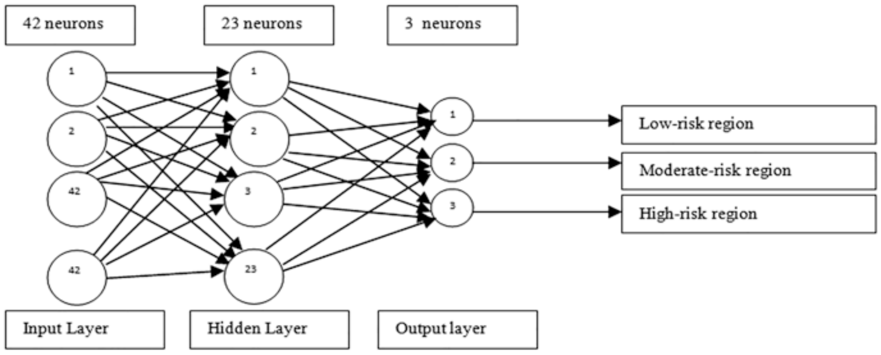


Fig. 1 The architecture of suggested deep learning network

In this work, each text word will be converted to a dummy variable. Thus, a text is represented as a numeric variable that represents a category.

3.2.2 Input Layer of ANN

This layer is used to feed in the input to the neural network. The data containing 42 input features is fed into the network using this. The perceptrons or nodes in the layer should ideally be equal to the features in use. The layer also consists of a bias unit 'x_o' which has the value of 1. ReLU is used as the activation function

$$\text{ReLu} = \text{R}(\text{z}) = \max(0, z) \tag{16}$$

where z = 0,1,2…0.42

3.2.3 Hidden Layer of ANN

The layer between the first layer (input) and last layer (output) and is responsible to process the input with the help of the activation function. Most problems can be solved by using only one layer of this type. The layer may also consist of bias unit. We have used 1 layer as a hidden layer, and the number of neurons is 23 to take output of the previous layer and using ReLU activation function to determine output

$$\text{ReLu} = \text{R}(\text{z}) = \max(0, z) \tag{17}$$

where z = 0,1,2…0.23

3.2.4 Output Layer of ANN

The output layer provides the output determined by the network. Every network has at least one neuron in the output layer. Three neurons are used in this layer each for low-, moderate- or high-risk region. The layer uses softmax as the activation function

$$\text{Softmax} = \sigma(\text{z})\text{I} = \frac{e_j^z}{\sum_{j=1}^{k} e_j^z} \tag{18}$$

where j = 0,1,2.

3.2.5 Activation Function

The function is used to map the values between 0 and 1 or − 1 and 1.The activation function can be of two types:

1. *Linear* – A function that is not confined within a range, such as f(x) = x
2. *Nonlinear* – A function which is confined within a range, such as sigmoid or tanh function

We have used ReLU (rectified linear unit) and softmax activation function explained in (16), (17) and (18).

3.2.6 Data Normalisation

Data normalisation requires a defined scale; the more accurate the chosen scale, the better it is. In this work, we withdraw the data of outbreak from the data centre. After data processing this data, we divide them as train set, cross-validation set and test set. Using scale function, the training dataset is first fitted by calculating the parameters to a given range and then transformed. The cross-validation set and test set are just transformed.

3.2.7 Training the Parameters for ANN-MDOP

In ANN-MDOP algorithm, the training parameters are defined as X (characteristic input) and Y (characteristic output). We use rmsprop method to train parameters and reach closure whether the region is at low, moderate or high risk. The updating rule of rmsprop is as follows

$$E\left[w^2\right]_t = \beta E\left[w^2\right]_{t-1} + (1-\beta)\left(\frac{\delta C}{\delta g}\right)^2 \tag{19}$$

$$\mathrm{g}_{\mathrm{t}} = \mathrm{g}_{\mathrm{t}-1} - \frac{\mathrm{n}}{\sqrt{\mathrm{E}\left[\mathrm{w}^2\right]_{\mathrm{t}}}}\frac{\delta C}{\delta g} \tag{20}$$

where E[w] = moving average of squared gradients, dC/dg = gradient of the cost function w.r.t the weight, n = rate of learning and beta = moving average parameter.

4 Results Obtained from Experiments

The performance of ANN-MDOP algorithm from several aspects is discussed in this section.

4.1 Effect of Neurons and Hidden Layer

In case of artificial neural network, first we need to confirm the number of neurons and also the number of layers that would stay hidden in the network. In this experiment, the selected neurons in each layer are 42 neurons in the first layer (input layer). These neurons are equal to the characteristics in the data, and three in the last layer (output layer) which is equal to classes in which the data needs to be classified. For the hidden layer, we first choose the number as three, and we halve the number of neurons in each layer with early stopping and dropout. Figure 2a, b illustrates the performance, where the number of neurons is 23 in the first hidden layer, 12 in the second hidden layer and 6 in the third hidden layer. For number of hidden layer as 1 with 23 neurons, the performance is illustrated in Fig. 2c, d with early stopping and dropout. The accuracy in Fig. 2d which contains the chosen number of neurons and layer, the loss and accuracy are 0.3073 and 0.8889 on training set and 0.5532 loss and 0.7500 accuracy on cross-validation set at 163th epoch. On the test set, the loss is 0.4135 and accuracy is 0.8611.

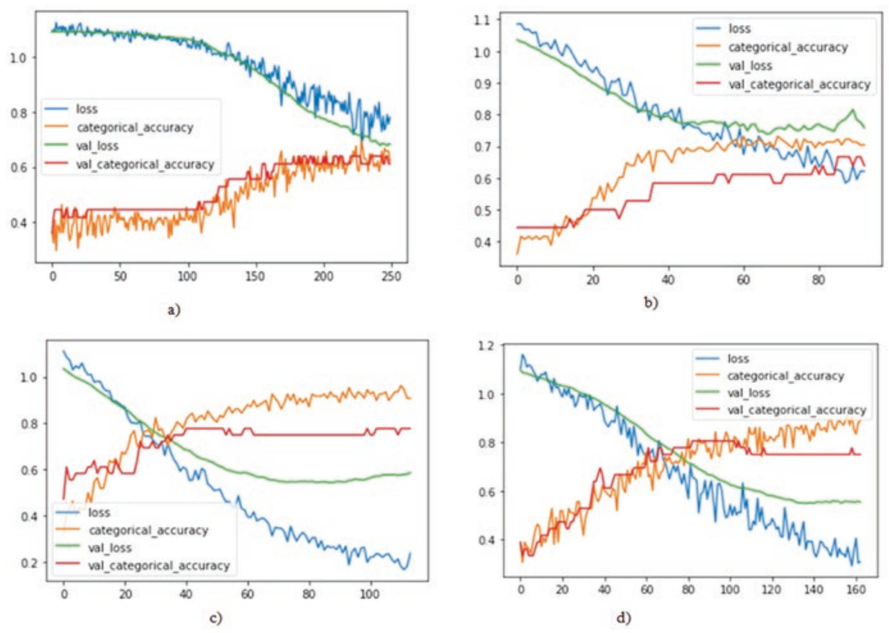


Fig. 2 (**a**) Performance when hidden layers are 3 with dropout rate of 0.5. (**b**) Performance when hidden layers are 3 with dropout rate of 0.2. (**c**) Performance when hidden layer is 1 with dropout rate of 0.2. (**d**) Performance when hidden layer is 1 with dropout rate of 0.5

4.2 Comparison of Dropout Rate

We compare the performance of ANN-MDOP algorithm using dropout rates of 0.2 and 0.5. Here, we set the same ANN iterations as 250 and train using early stopping and dropout to prevent overfitting. The dropout rates of 0.2 and 0.5 are compared in Fig. 2. The figure also shows the effect of different dropout rates on model performance [13]. The algorithm attains the precision of 0.5 on the chosen specifications.

4.3 Iteration Effect

We illustrate the training loss and accuracy with the iterations. As given in Fig. 2, the increase in neurons, hidden layer and dropout rate has a great impact on the number of iterations on which the model converges; the loss of the ANN-MDOP algorithm decreases, while accuracy increases. In Fig. 2a the number of iterations is 250 when the number of hidden layer is 3 with dropout rate of 0.5. Similarly, Fig. 2b shows the iterations are 87, when hidden layers are 3 with dropout rate of 0.2. In Figure 2c, d the iterations are 108 and 163 when the number of hidden layer is 1 with 23 neurons, and dropout rates are 0.2 and 0.5, respectively [14]. Thus the training with specifications in Fig. 2d is computationally cheaper with desired target of obtaining less loss and more accuracy.

5 Analysis of Results

Here, we provide the analysis of overall results about P&W-data.

5.1 Positive Case and Weather Data (P&W-Data)

As discussed above, we provide the loss, accuracy, precision measure and learning curve plotted under ANN-MDOP (P&W-data) algorithm. In this work, the selected number of neurons is 42 in input layer and 3 in output layer. Also, 1 hidden layer is considered with 23 neurons. The number of iterations is 250. However, the algorithm runs 163 times and converges due to early stopping and dropout. Figure 2d shows the chosen number of neurons and layer and the loss and accuracy as 0.3073 and 0.8889 on training set and 0.5532 loss and 0.7500 accuracy on cross-validation set at 163th epoch. On the test set, the loss is 0.4135 and accuracy is 0.8611. Thus, we can draw the conclusion that the performance of ANN-MDOP (P&W-data) is best at chosen specifications to predict the outbreak. In conclusion, the accuracy of outbreak prediction depends on the contrasting features, i.e. the lower the correlation

[15] between the features, the higher is the chance of accurate predictions. For some diseases at low risk, only a few characteristics can get a good prediction of the outbreak. But for a complex disease, such as chikungunya, malaria and dengue, only few features of highly correlated data are incorrect way to predict the outbreak. As identified from Fig. 2d, the algorithm is moderately accurate, which is approximately 80%. Therefore, in this work, we grasp not only the positive case statistical data but the weather condition data of regions also. Hence, we infer that by combining these two datasets with more features, the rate of accuracy can further be improved, for better evaluation of the prediction of outbreak of disease due to a vector [16].

6 Conclusion

In this work, a new artificial neural network-based multimodal outbreak prediction (ANN-MDOP) algorithm is proposed with the use of contrasting data. To the finest of our understanding, none of the existing works have focused on contrasting data in the area of analysis of medical data. The prediction accuracy of our suggested ANN algorithm is 86%.

References

1. K. Usha Rani: Analysis Of Heart Diseases Dataset Using Neural Network Approach. International Journal of Data Mining & Knowledge Management Process (IJDKP) Vol.1, No.5, September 2011
2. Shraddha Shivhare and Rajesh Shrivastava: Automatic Bone Marrow White Blood Cell Classification using Morphological Granulometric Feature of Nucleus. Oriental Journal Of Computer Science & Technology ISSN: 0974-6471 No. (1) Vol. 5 June 2012
3. Md. Osman Goni Nayeem, Maung Ning Wan, et al.: Prediction of Disease Level Using Multilayer Perceptron of Artificial Neural Network for Patient Monitoring International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-5 Issue-4, September 2015
4. Sivaranjani.R: Artificial Intelligence Model for Earlier Prediction of Cardiac Functionalities Using Multilayer Perceptron. International Conference on Physics and Photonics Processes in Nano Sciences Journal of Physics: Conference Series 1362 (2019)
5. Marios Anthimopoulos, Stergios Christodoulidis, et al.: Lung Pattern Classification for Interstitial Lung Diseases Using a Deep Convolutional Neural Network. IEEE Transactions On Medical Imaging, VOL. 35, NO. 5, MAY 2016
6. Rahib H. Abiyev and Mohammad Khaleel Sallam Ma'aitah: Deep Convolutional Neural Networks for Chest Diseases Detection. Journal of Healthcare Engineering Volume 2018
7. Agus Qomaruddin Munir and Edi Winarko: Classification Model Disease Risk Areas Endemicity Dengue Fever Outbreak Based Prediction Of Patients, Death, IR and CFR Using Forecasting Techniques. International Journal of Computer Applications, 2015
8. Juan M. Scavuzzoa, Francisco Truccoa et al.: Modeling Dengue Vector Population Using Remotely Sensed Data and Machine Learning (Preprint submitted to Acta Tropica, 2018)

9. B. Mahalakshmi and G. Suseendran: Prediction of Zika Virus by Multilayer Perceptron Neural Network (MLPNN) Using Cloud. International Journal of Recent Technology and Engineering (IJRTE), Volume-8, September 2019
10. V. Janani, N. Maadhuryaa, D. Pavithra and S. Ramya Sree: Dengue Prediction Using Multilayer Perceptron – A Machine Learning Approach. International Journal of Research in Engineering, Science and Management Volume-3, Issue-3, March-2020
11. Oladimeji Mudele, Fábio M. Bayer et al.: Modeling The Temporal Population Distribution Of Ae. Aegypti Mosquito Using Big Earth Observation Data (pre-print) 2019
12. Vijeta Sharma, Ajai Kumar et al.: Malaria Outbreak Prediction Model Using Machine Learning. International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume 4 Issue 12, December 2015
13. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
14. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor Flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
15. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3
16. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Regression, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_4

Eukaryotic Plasma Cholesterol Prediction from Human GPCRs Using K-Means with Support Vector Machine

Ramamani Tripathy and Rudra Kalyan Nayak

1 Introduction

1.1 Definition of Cell Membrane

All living organisms like human being for instance basically includes trillions of countable cells. In living things different cells have diversified functionalities. They are the basic building block of all organisms. The important types of cells are generally utilized in human body such as nerve cells, muscle cells, smooth cells, red blood cells, stem cells, etc. Cell membrane is treated as the structure of all cells which is also known as wall, and it creates a boundary between external leaflets and internal leaflets. Cell membrane is responsible for controlling the movement of every substance within and outside the cell. The alternative name of cell membrane is plasma membrane. It is a phospholipid bilayer structure, and its main work is to environ the cell. Plasma membrane has many responsibilities such as selectively giving permission to some substances into the cell and maintaining the barrier between the external and internal environment. Another major role is its cell signaling and communication among different cells [1–5].

R. Tripathy
Department of Master of Computer Application, United School of Business Management, Bhubaneswar, Odisha, India

R. K. Nayak (✉)
Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation (Deemed to be University), Guntur, Andhra Pradesh, India

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_10

1.2 Components of Cell Membrane

Generally plasma membranes are categorized into four different fractions: phospholipid bilayer, cholesterol, carbohydrates, and proteins.

1.2.1 Phospholipid Bilayer

We know that all plasma membranes are composed of phospholipid bilayer. This bilayer is made up of two fatty acid tails which are hydrophobic in nature and one phosphate group head which is hydrophilic in nature. This implies that the group head is attracted to water, whereas the fatty acid tail is repelled by water (Fig. 1).

1.2.2 Carbohydrates

Carbohydrate is another component of cell membrane, and it is basically found on the external leaflet of cells. These carbohydrate chains may consist of 2–60 monosaccharide units and can be either straight or branched. The structure of carbohydrate is composed of sugar molecules, and it includes both glucose and fructose, which are monosaccharaides. Complex carbohydrates are starches found in foods like wheat, potato, and beans and are often made up of a large number of sugar molecules bound together [2–6].

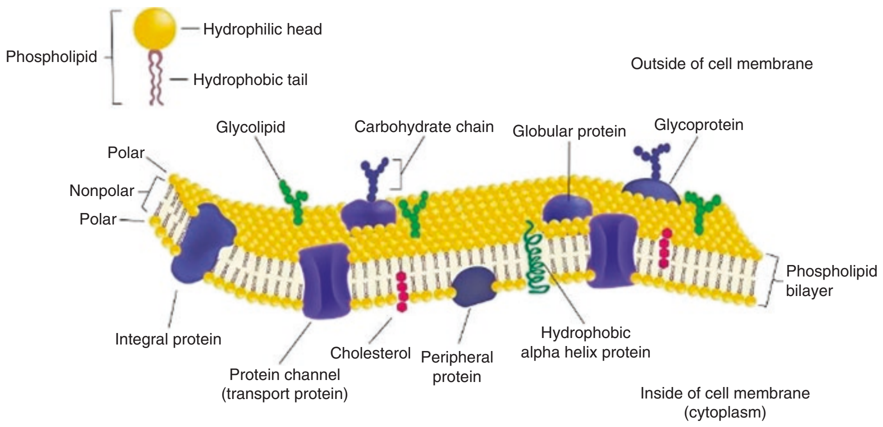


Fig. 1 Structure of cell membrane and its components. (https://biologydictionary.net/cell-membrane)

1.2.3 Proteins

Another most important component of cell membrane is protein. Two types of proteins are there in cell membrane; one is peripheral protein, and another is integral protein. On both inside and outside leaflet of the membrane, peripheral proteins are found which are loosely present with the hydrophilic surfaces of the lipid bilayer. It is a temporary protein and can be detached from the membrane without disrupting it through application of polar reagents. But integral membranes are found within the membrane and it resides without water region. The most important integral protein is transmembrane protein which has multi-pass property from inside to outside of the membrane. Transmembrane protein is a long amino acid chain, so it is multi-passes in nature [1–6].

1.2.4 Cholesterol

In addition with other components, the human body has another important component, that is, cholesterol. Cholesterol is waxy fatlike substances. It is not uniformly distributed among cell membranes and is hydrophobic in nature. It decreases the permeability of lipid membranes which means it restricts the molecules from entering into the cell membrane. The main function of cholesterol is vitamin D production, bile secretion, and biosynthesis of steroid hormones (Fig. 2).

The structure of cholesterol is made up of four fused rings which are named as A, B, C, and D and is attached with carbon numbered in the sequences in addition to an eight-numbered and branched hydrocarbon chain attached to the D ring.

1.3 G-Protein-Coupled Receptor

G-protein-coupled receptor (GPCR) is the largest receptor family of plasma membrane. Generally, different numbers of receptors are located in mammalian cells, and these receptors are known as proteins which have the capability to bind specific molecules. The binding molecules are treated as ligand, and it may be counted as any molecules from inorganic minerals to organism-created proteins, hormones,

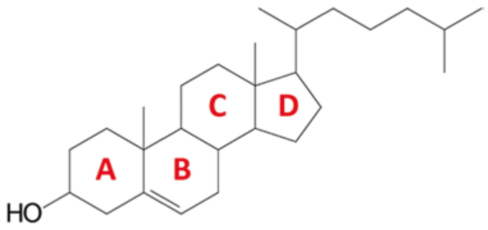


Fig. 2 Structure of cholesterol. (https://themedicalbiochemistrypage.org/cholesterol.php)

and neurotransmitters. As GPCR proteins have long strain of amino acids, it passes seven times within cell membrane. Probably, above 820 genes are available in GPCR family. Here, there exist so many subfamilies. All proteins have different sequences and functionalities. GPCR protein family has numerous subfamilies, such as Class A rhodopsin-like, Class B secretin-like, Class C metabotropic glutamate/pheromone, Class F frizzled (FZD), taste receptors (TAS1R, TAS2R), vomeronasal receptors (VN1R, VN2R), and 7TM orphan receptors [1–10].

We know that GPCR receptor is included under integral membrane proteins that possess seven transmembrane helices. These structures indicate how ligand binding at the extracellular side of a receptor leads to conformational changes in the cytoplasmic side of the receptor. In Fig. 3 seven helices are present. Each time, a helix enters into the membrane and exits from them. Each helix has dissimilar amino acid sequences in their transmembrane region. The total span of helix is starting from N-terminus to C-terminus. GPCRs play a precise job in the case of human drug discovery. According to cell biology, all components of plasma membrane are important, but cholesterol has extra key feature on the membrane. All types of detection can be done by cholesterol in the human body; in addition it has a modulatory role and ligand-binding property of GPCRs. Cholesterol is a key constituent of human cell membranes and in addition a proven modulatory role on the function and ligand-binding properties of GPCRs [5–18] (Fig. 4).

As days go by, so many researchers have their studies on cellular cholesterol target on membrane receptor like GPCR. Numerous approaches were used for prediction purpose. Most probably the techniques are on machine learning, soft computing, data mining, etc. Clustering is nothing but grouping of similar objects in one cluster point. There are numerous algorithms that have been utilized for clustering the same data points. Among all K-means is an important algorithm which is used

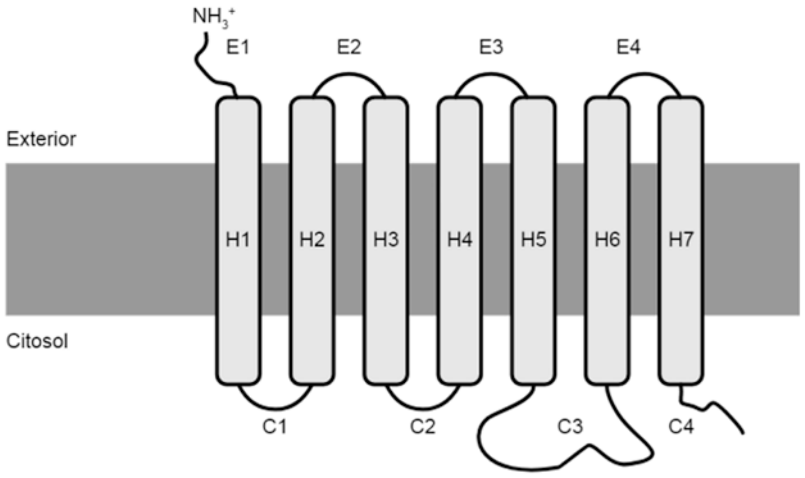


Fig. 3 Structure of GPCR receptor. (https://upload.wikimedia.org/wikipedia/commons/1/12/GPCR_structure_and_receptor.svg)

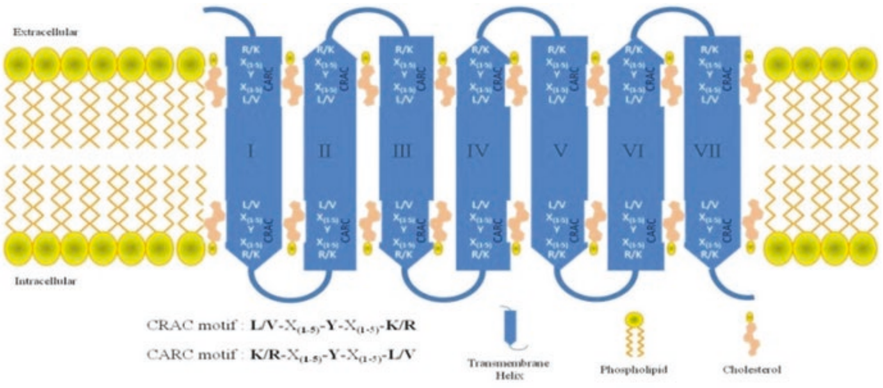


Fig. 4 Cholesterol with seven helices of GPCR protein target sites

when you have unlabeled data. Support vector machine (SVM) plays a significant role for prediction and classification among dissimilar objects. For that purpose we have developed a hybrid approach K-means with support vector machine for prediction of membrane protein GPCR with membrane cholesterol [17–20]. Here we have explored a hybrid approach based on K-means and support vector machine. From analysis we show how plasma membrane cholesterol targets on helical sites of membrane GPCR receptor and finds out the valid domain sites. We will discuss our manuscript through proposed methods in addition with experimental analysis and conclusion.

2 Flow of Work Elaboration (Fig. 5)

Step 1: All data sets of receptor protein was collected from UniProt [21] database. Generally the helical data has different sequences according with their transmembrane span. Totally seven helices are present in membrane receptor of GPCR. Each file has included above 900 proteins.

Step 2: Next to cholesterol, dictionary is created using sliding window concept.

$$\mathcal{D} = \left\{ d_5, d_6, d_7, d_8, d_{9,} d_{10}, d_{11}, d_{12}, d_{13} \right\}$$

Using (L/V-$\boldsymbol{X}_{(1-5)}$-Y-$\boldsymbol{X}_{(1-5)}$-R/K) forward (CRAC) and (R/K-$\boldsymbol{X}_{(1-5)}$-Y-$\boldsymbol{X}_{(1-5)}$-L/V) backward (CARC) algorithms, we set the cellular membrane lipid dictionary.

Step 3: After setting all data set, we separated them according with motif types such as 11,12 13,14,15,21,22,23,24,25,31,32,33,34,35,41,42,43,44,45,51,52,53,54,5 5 in addition with forward and backward motif sequences.

Step 4: Moving to next we found so many cluster points using K-means algorithm. Then we classify the cluster points applying support vector machine approach.

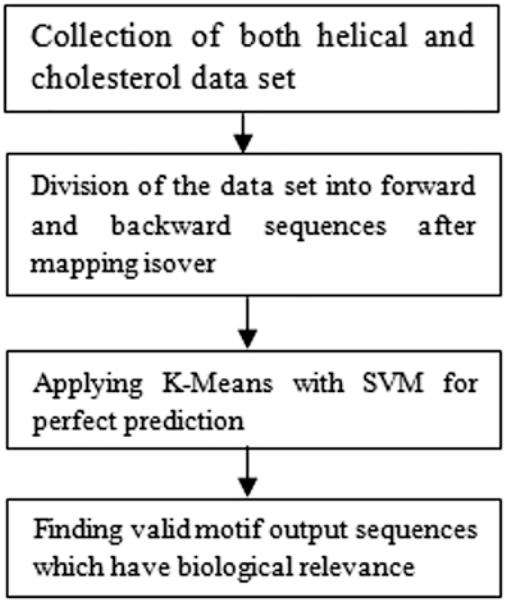

Collection of both helical and cholesterol data set
Division of the data set into forward and backward sequences after mapping isover
Applying K-Means with SVM for perfect prediction
Finding valid motif output sequences which have biological relevance

Fig. 5 Flow of our proposed work

Table 1 Forward cholesterol motif sequences observed from GPCR proteins

Motif-Length	L-$X_{(1\text{-}5)}$-Y-$X_{(1\text{-}5)}$-R and L-$X_{(1\text{-}5)}$-Y-$X_{(1\text{-}5)}$-K	V-$X_{(1\text{-}5)}$-Y-$X_{(1\text{-}5)}$-R and V-$X_{(1\text{-}5)}$-Y-$X_{(1\text{-}5)}$-K	Sum
5	69	60	129
6	420	220	640
7	103	124	227
8	470	322	792
9	717	240	957
10	240	128	368
11	161	320	481
12	104	162	266
13	30	65	95
Total	**2314**	**1641**	**3955**

Step 5: In the final step, we found valid motif sequences with their weight and amino acid priority which have biological relevance for all drug designers.

From Tables 1 and 2, we found the total mapping motif sequences for both CRAC and CARC. In Table 1 total forward motif is 3955, and for reverse part total motif was found to be 1726 which is mentioned in Table 2.

Table 2 Backward cholesterol motif sequences observed from GPCR proteins

Motif-Length	R-$X_{(1-5)}$-Y-$X_{(1-5)}$-L and R-$X_{(1-5)}$-Y-$X_{(1-5)}$-V	K-$X_{(1-5)}$-Y-$X_{(1-5)}$-L and K-$X_{(1-5)}$-Y-$X_{(1-5)}$-V	Sum
5	43	28	71
6	77	58	135
7	101	71	172
8	125	75	200
9	225	193	418
10	210	205	415
11	96	90	186
12	47	48	95
13	120	94	214
Total	**1044**	**862**	**1726**

3 Methodology Discussion

In recent days, clustering technique is considered as one of the most frequent investigative data analysis approaches which has been used to obtain the proper data set structure. Also it has the responsibility to recognize groups having similarity and dissimilarity among the data points. So many cluster algorithms are there. Among all, k-means algorithm is one by employing which all data sets are divided into K predefined dissimilar subparts without overlie whereas each data point fits in to single group [22–25].

3.1 K-Means Clustering

Step 1. Identify cluster point K.
Step 2. Select centroids according with the data points, and after that arbitrarily choose *K* data points for the centroids with no substitute.
Step 3. Maintain iterating till no modifications arise to the centroids.
Step 4. Calculate the sum of the squared distances among data points with every centroid.
Step 5. Then put every data part to the cluster to which one is closest to.
Step 6. By calculating the centroid for each cluster of all data parts final valid motif sequences are found.

The objective function is

$$W = \sum_{n}^{i=1} \sum_{K}^{k=1} x_{ik}\, y^{i} - \mu_{k}^{\;2} \tag{1}$$

Anywhere $x_{ik} = 1$ for each data part y_i if it belongs to cluster k; or else, $x_{ik} = 0$. As well, μ_k is treated as centroid of y_i’s cluster.

This minimization dilemma has two fractions. Here first we minimize W w.r.t. x_{ik} with μ_k permanent. Next, we minimize W w.r.t. μ_k with treat x_{ik} permanent. Finally, we differentiate W w.r.t. x_{ik} first plus bring up-to-date cluster assignments (E-step). After that, we differentiate W w.r.t. μ_k plus recalculate the centroids following the cluster assignments from the earlier step (M-step). As a result, E-step is

$$\begin{gathered}\frac{\partial W}{\partial x_{ik}} = \sum_{n}^{i=1} \sum_{K}^{k=1} x_{ik} \left\| y^i - \mu_k \right\|^2 \\ => x_{ik} = \{ \begin{matrix} 1, \textit{if } k = \operatorname{argmin}_j \left\| y^i - \mu_j \right\|^2 \\ 0, \text{ otherwise} \end{matrix} \end{gathered} \tag{2}$$

Another way allocates the computed data parts y_i to the nearby cluster judged with its summation of squared distance from the cluster's centroid.

Also M-step is

$$\begin{gathered}\frac{\partial W}{\partial \mu_k} = 2 \sum_{m}^{i=1} x_{ik} \left(y^i - \mu_k \right) = 0 \\ => \mu_k = \frac{\sum_{i=1}^{n} x_{ik} y^i}{\sum_{i=1}^{n} x_{ik}} \end{gathered} \tag{3}$$

whichever converts to recalculate the centroid of each one cluster to mirror the new assignments.

3.2 Support Vector Machine

Support vector machine was introduced in COLT-92 by Boser, Guyon, and Vapnik. It is now a very well-accepted algorithm which has been proposed from statistical learning theory. As this algorithm has been used for decades and also performs well, many researchers applied this in many new emerging fields like image recognition, data classification, micro-array gene expression, and bioinformatics. SVM is a supervised learning algorithm, and its resultant depends on training with testing data set. Basically SVM approach uses kernel trick technique to transform the given data, and after that it obtains an optimal boundary amid the possible outputs on the basis of transformations [26–30].

A training set that includes label pairs (j_i, v_i), $i = 1, \ldots, n$ everywhere $w_i \in \Re^n$ and $v \in \{\textit{class label}\}^i$, the results of SVM approach are obtained using optimization problem as given below:

$$\min_{x,y,\xi} \frac{1}{2} W^M u + B \sum_{n}^{i=1} \xi_i \tag{4}$$

$$\text{Subject to}: v_i(z^M\Phi(j)+y \geq 1-\xi_i, \xi_i \geq 0. \tag{5}$$

In Eq. (6) decision function is denoted:

$$R = pfn\left(\sum_{n}^{i=1} x_i\alpha_i A(x_i, x) + \rho\right) \tag{6}$$

Training vector j_i is mapped into higher-dimensional space with the help of kernel function ϕ. In case of error, $D > 0$ is also treated as penalty structure. Accordingto SVM theory, every time with the help of hyperplanes, data points are computed, though it is not suitable for finding linear solution in 2D spaces. Therefore, we solve it through kernel function $k(u_i, u_y) \equiv \Phi(u_i)^N\Phi(u_y)$ for multidimensional data. Utilizing divergent kernel functions, in below Eqs. (7) and (8) as well as in (9) equation all steps of SVM approach are elaborated.

$$\text{Linear kernel}: (u_i, u_y) = u_i^N u_y \tag{7}$$

$$\text{Polynomial kernel}: \ f(u_i, u_y) = (\gamma u_i^N u_y + r)^e, \gamma > 0, \text{and} \tag{8}$$

$$\text{Radial basis kernel (RBF)}: \ (u_i, u_y) = \exp(-\gamma \| u_i - u_y \|^2), \gamma \tag{9}$$

On the basis of training data size, all kernel parameters are pretentious.

4 Experimental and Result Analysis

The paper is computed based on Intel i5 processor with 8GB hard disk plus Windows 10 operating system for concluding the test, and the whole resultant instruction is written by Python 3. To calculate our experimental analysis, we explored hybrid algorithm and many more concepts. Here we have depicted intricately our whole work flow using the following phases:

Phase 1: In the beginning phase, we have retrieved all protein data sets from UniProt database according with helical name and in addition with computed cholesterol dictionary.

Phase 2: Once filtration process is over amid both data sets we separate the sequence residue which starts from L/V or R/K for forward motif and R/K or L/V for backward sequences.

Phase 3: After gathering both data set motif sequences, we put sliding window concept on both CRAC and CARC, and window size may vary from 5, 6, 7, 8, 9, 10, 11, 12, to 13. This cholesterol dictionary can be calculated using CRAC (L/V-$X_{(1–5)}$-Y-$X_{(1–5)}$-R/K) and CARC (R/K-$X_{(1–5)}$-Y/F-$X_{(1–5)}$-L/V).

Phase 4: After completion of phase 3 work, we go for next step where we have implemented our planned algorithm *K*-means and support vector machine for prediction of membrane cholesterol with membrane receptor GPCR.

K-means algorithm is one by applying which all data sets are divided into K predefined dissimilar subparts without overlie whereas each one data point fits in to single group [31].

In this algorithm the centroids are recomputed in recurring manner using all data points assigned to that centroid's cluster. Each centroid defines one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance. More formally, if c_i is the collection of centroids in set C, then each data point x is assigned to a cluster based on

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i \tag{10}$$

Also we used another algorithm which is required for prediction purpose. SVM is a supervised learning algorithm, and its resultant depends on training with testing data set. Basically SVM approach uses kernel trick technique to transform the given data, and after that it obtains an optimal boundary amid the possible outputs on the basis of transformations.

SVM approach is obtained using optimization problem as given below:

$$\min_{x,y,\xi} \frac{1}{2} W^M u + B \sum_{n}^{i=1} \xi_i \tag{11}$$

Table 3 shows prediction output of plasma membrane cholesterol from membrane receptor for forward region, and Table 4 shows prediction output of plasma membrane cholesterol from membrane receptor for backward region. From our experiment we conclude that CARC motif has more target region than CRAC motif. Both CARC and CRAC targeted all helical sites and gave well prediction results which have biological importance. The higher the motif type, the greater the target sites found in CARC region in plasma membrane of mammalian cells [32, 33].

5 Conclusion

Day by day research on mammalian biology has been a challenging factor for all scientists. Due to changing environment, several unknown diseases are visible in the human body. To solve these new problems, different techniques are adopted in current scenario. In the human body, plasma membrane cholesterol takes a major role. From literature it is revealed that automatically cholesterol is formed in the body, but sometimes that amount is not sufficient. Another way is there through which body can get cholesterol from outsource things. Generally there are two types of

Table 3 Prediction output of plasma membrane cholesterol from membrane receptor for forward region

Helix	Sequence and motif type	Name of ID
Helix 5	LIVGFCYVRIWTK (55)	Q13585
Helix 5	LAIISIYYYHIAK (55)	P28336
Helix 5	LVILLSYVRVSVK (55)	P49683
Helix 5	LVICLCYLLIVVK (55)	P32745
Helix 5	LAMSFCYLVIIR (54)	P32248
Helix 5	LIMLFCYGFTLR (54)	P25025
Helix 5	LCLSILYGLIGR (54)	O43193
Helix 5	LVISVCYSLMIR (54)	P41146
Helix 5	LLTLAAYGALGR (54)	Q96G91
Helix 5	LLMLVLYGRIFR (54)	P08908
Helix 5	LIMGVCYFITAR (54)	Q9NPB9
Helix 5	VLGLTYARTLR (44)	O43603
Helix 5	VMAVAYGLISR (44)	P32239
Helix 5	VTCTVYAIKTR (44)	Q14833
Helix 7	VVNPIIYSYK (52)	Q9UBY5
Helix 7	VLNPIVYSVK (52)	Q8NH63
Helix 7	VSLGMLYMPK (52)	Q14833
Helix 7	VSLGMLYVPK (52)	O15303
Helix 7	VALGMLYMPK (52)	Q14831
Helix 7	VSLGMLYMPK (52)	O00222
Helix 7	VVNPIVYAYR (52)	P29275
Helix 7	LDPVLYFLAGQR (45)	P41231
Helix 7	LNPLVYVIVGKR (45)	P30411

cholesterols found in the human body, namely, bad cholesterol and good cholesterol. We also know that membrane cholesterols are modulated by membrane proteins. Here we focus on transmembrane sites of GPCR proteins, and in addition cholesterol targets on those helical sites within the cell membrane. All the time membrane cholesterol targets on each helix inside the cell membrane which causes many target sequences generation. Likewise target sites are generated, and then we applied our approach *K*-means with support vector machine for prediction. From this analysis we found that some subfamilies have a preference for forward motif, some for reverse motif, and some for both. Among forward and backward sequences, here more number of reverse motifs are observed which have biological relevance. Hence, we wind up the projected hybrid algorithm is also proficient to forecast the CRAC/CARC motif sequences with high reliability, and it gave immense implications in medicine and molecular targeting domains.

Table 4 Prediction output of plasma membrane cholesterol from membrane receptor for backward region

Helix	Sequence and motif types	Name of ID
Helix 2	KSVTDIYLLNLAL (55)	P49238
Helix 2	KTATNIYIFNLAL (55)	P41143
Helix 2	KTATNIYIFNLAL (55)	P41145
Helix 2	KTATNIYIFNLAL(55)	P41146
Helix 2	KHPAVIYMANLAL(55)	P55085
Helix 2	KFHNRMYFFIGNL(55)	Q99500
Helix 2	KRVENIYLLNLAV(55)	O00421
Helix 2	KSAFQVYMINLAV(55)	Q9Y271
Helix 2	KTVPDIYICNLAV(55)	Q969V1
Helix 2	KTPTNYYLFSLAV(55)	Q9GZQ4
Helix 2	KTATNIYLLNLAV(55)	P31391
Helix 2	KKVSSIYIFNLAV(55)	P50052
Helix 2	RTTTNLYLGSMAV(55)	O43193
Helix 2	RTPTNYYLFSLAV(55)	Q9HB89
Helix 2	KLTVPRFLMCNL (54)	P23945
Helix 2	KLTVPRFLMCNL(54)	P22888
Helix 2	KLNVPRFLMCNL(54)	P16473
Helix 2	KKSRMTFFVTQL(54)	Q6W5P4
Helix 2	KHSRLFFFMKHL(54)	P30559
Helix 2	KLSTIGFILTGL(54)	Q9NYW0
Helix 2	RPMYYFIGNLAL (45)	P21453
Helix 2	RALSVFIKDAAL (45)	P43220
Helix 2	RQPSNYLIV (42)	P34969
Helix 2	RTVTNYFLV (42)	P29371
Helix 2	RTVTNYFIV (42)	O43613
Helix 2	RTVTNYFIV (42)	O43614
Helix 2	RRWVYYCLV (42)	O95977
Helix 2	RTVTNYFIV (42)	P21452
Helix 2	RTVTNYFLV (42)	P25103
Helix 2	RTPTNYFIV (42)	P35368
Helix 2	RQPLNYILV (42)	P03999
Helix 3	RHHWVFGVL (42)	Q86VZ1
Helix 3	RYLSIFWVL (42)	Q9NYW4
Helix 3	KMSFFSGMLLL (35)	P32248
Helix 3	KEVNFYSGILL (35)	P25024
Helix 3	KLQRFIFHVNL (35)	P47900
Helix 3	KLVRFLFYTNL (35)	P41231
Helix 3	KFVRFLFYWNL (35)	P51582
Helix 3	KIANFSNYIFL (35)	Q9NYW0
Helix 6	RTVKLIFAIV (52)	P46094
Helix 6	RAMRVIFAVV (52)	P25024
Helix 6	RAMRVIFAVV (52)	P25025
Helix 6	RTVKMTFIIV (52)	P30559
Helix 6	RTVKMTFVIV (52)	P37288
Helix 6	RTVKMTFVIV (52)	P47901

Table 4 (continued)

Helix	Sequence and motif types	Name of ID
Helix 6	RIHIFWLL (32)	P49019
Helix 6	RRRTFCLL (32)	P49683
Helix 6	RSALFQIL (32)	O14514
Helix 6	RIGVFSVL (32)	Q9UP38
Helix 6	RIGLFSVL (32)	Q9ULW2
Helix 6	RIGVFSVL (32)	Q14332
Helix 6	RIGVFSVL (32)	O75084
Helix 6	RLGLFTVL (32)	Q9H461

References

1. Tripathy, R., Mishra, D., Konkimalla, V.B., Nayak, R.K.: A computational approach for mining cholesterol and their potential target against GPCR seven helices based on spectral clustering and fuzzy c-means algorithms. J. Intell. Fuzzy Syst. **35**(1), 305–314 (2018)
2. Ahmed, M.R., Zhan, X., Song, X., Kook, S., Gurevich, V.V., Gurevich, E.V.: Ubiquitin ligase parkin promotes Mdm2–Arrestin interaction but inhibits arrestin ubiquitination. Biochemistry. **50**(18), 3749–3763 (2011)
3. Alvarez-Curto, E., Inoue, A., Jenkins, L., Raihan, S.Z., Prihandoko, R., Tobin, A.B., Milligan, G.: Targeted elimination of G proteins and arrestins defines their specific contributions to both intensity and duration of G protein-coupled receptor signaling. J. Biol. Chem. **291**(53), 27147–27159 (2016)
4. Tripathy, R., Mishra, D., Konkimalla, V.B.: A novel fuzzy C-means approach for uncovering cholesterol consensus motif from human G-protein coupled receptors (GPCR). Karbala Int. J. Modern Sci. **1**(4), 212–224 (2015)
5. Attramadal, H., Arriza, J.L., Aoki, C., Dawson, T.M., Codina, J., Kwatra, M.M., et al.: Beta-arrestin2, a novel member of the arrestin/beta-arrestin gene family. J. Biol. Chem. **267**(25), 17882–17890 (1992)
6. Baameur, F., Morgan, D.H., Yao, H., Tran, T.M., Hammitt, R.A., Sabui, S., et al.: Role for the regulator of G-protein signaling homology domain of G protein-coupled receptor kinases 5 and 6 in β2-adrenergic receptor and rhodopsin phosphorylation. Mol. Pharmacol. **77**(3), 405–415 (2010)
7. Baylor, D.A., Lamb, T.D., Yau, K.W.: Responses of retinal rods to single photons. J. Physiol. **288**(1), 613–634 (1979)
8. Benovic, J.L., DeBlasi, A., Stone, W.C., Caron, M.G., Lefkowitz, R.J.: Beta-adrenergic receptor kinase: primary structure delineates a multigene family. Science. **246**(4927), 235–240 (1989)
9. Benovic, J.L., Kühn, H., Weyand, I., Codina, J., Caron, M.G., Lefkowitz, R.J.: Functional desensitization of the isolated beta-adrenergic receptor by the beta-adrenergic receptor kinase: potential role of an analog of the retinal protein arrestin (48-kDa protein). Proc. Natl. Acad. Sci. **84**(24), 8879–8882 (1987)
10. Benskey, M.J., Perez, R.G., Manfredsson, F.P.: The contribution of alpha synuclein to neuronal survival and function–implications for Parkinson's disease. J. Neurochem. **137**(3), 331–359 (2016)
11. Carman, C.V., Parent, J.L., Day, P.W., Pronin, A.N., Sternweis, P.M., Wedegaertner, P.B., et al.: Selective regulation of Gαq/11 by an RGS domain in the G protein-coupled receptor kinase, GRK2. J. Biol. Chem. **274**(48), 34483–34492 (1999)
12. Bockaert, J., Pin, J.P.: Molecular tinkering of G protein-coupled receptors: an evolutionary success. EMBO J. **18**(7), 1723–1729 (1999)

13. Gimpl, G.: Interaction of G protein coupled receptors and cholesterol. Chem. Phys. Lipids. **199**, 61–73 (2016)
14. Greenwood, A.I., Tristram-Nagle, S., Nagle, J.F.: Partial molecular volumes of lipids and cholesterol. Chem. Phys. Lipids. **143**(1–2), 1–10 (2006)
15. Jafurulla, M., Tiwari, S., Chattopadhyay, A.: Identification of cholesterol recognition amino acid consensus (CRAC) motif in G-protein coupled receptors. Biochem. Biophys. Res. Commun. **404**(1), 569–573 (2011)
16. Oates, J., Watts, A.: Uncovering the intimate relationship between lipids, cholesterol and GPCR activation. Curr. Opin. Struct. Biol. **21**(6), 802–807 (2011)
17. Oddi, S., Dainese, E., Fezza, F., Lanuti, M., Barcaroli, D., De Laurenzi, V., Maccarrone, M.: Functional characterization of putative cholesterol binding sequence (CRAC) in human type-1 cannabinoid receptor. J. Neurochem. **116**(5), 858–865 (2011)
18. Sun, X., Whittaker, G.R.: Role for influenza virus envelope cholesterol in virus entry and infection. J. Virol. **77**(23), 12543–12551 (2003)
19. Epand, R.M., Thomas, A., Brasseur, R., Epand, R.F.: Cholesterol interaction with proteins that partition into membrane domains: an overview. In: Cholesterol Binding and Cholesterol Transport Proteins, pp. 253–278. Springer, Dordrecht (2010)
20. Hamouda, A.K., Chiara, D.C., Sauls, D., Cohen, J.B., Blanton, M.P.: Cholesterol interacts with transmembrane α-helices M1, M3, and M4 of the Torpedo nicotinic acetylcholine receptor: photolabeling studies using [3H] azicholesterol. Biochemistry. **45**(3), 976–986 (2006)
21. UniProt Consortium: The universal protein resource (UniProt). Nucleic Acids Res. **36**(suppl_1), D190–D195 (2007)
22. Fahim, A.M., Salem, A.M., Torkey, F.A., Ramadan, M.A.: An efficient enhanced k-means clustering algorithm. J. Zhejiang Univ. Sci. A. **7**(10), 1626–1633 (2006)
23. Naik, P.P.S., & Gopal, T.V.: Segmentation of Magnetic Resonance Brain Tumor Using Integrated Fuzzy KMeans Clustering
24. Fatma, M., Sharma, J.: Leukemia image segmentation using K-means clustering and HSI color image segmentation. Int. J. Comput. Appl. **94**(12), 6–9 (2014)
25. Xu, R., Wunsch, D.: Survey of clustering algorithms. IEEE Trans. Neural Netw. **16**(3), 645–678 (2005)
26. Cortes, C.: WSupport-vector network. Mach. Learn. **20**, 1–25 (1995)
27. Zhang, X., Zheng, X.: Comparison of text sentiment analysis based on machine learning. In: 2016 15th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 230–233. IEEE (2016, July)
28. Upadhyay, V.P., Panwar, S., Merugu, R., Panchariya, R.: Forecasting stock market movements using various kernel functions in support vector machine. In: Proceedings of the International Conference on Advances in Information Communication Technology & Computing, pp. 1–5 (2016, August)
29. Nayak, R.K., Mishra, D., Rath, A.K.: An optimized SVM-k-NN currency exchange forecasting model for Indian currency market. Neural Comput. & Applic. **31**(7), 2995–3021 (2019)
30. Chandra, N.: Support vector machine classifier for predicting drug binding to P-glycoprotein. J. Proteomics Bioinform. **2**, 193–201 (2009)
31. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
32. JHA, A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
33. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3

Dr. Rudra Kalyan Nayak is presently working as Associate Professor in the Department of Computer Science and Engineering at Koneru Lakshmaiah Education Foundation (deemed to be University), Green Fields, Vaddeswaram, Andhra Pradesh, India. He has got his M.Tech in Information Technology and Ph.D in Computer Science and Engineering from Siksha ‘O’ Anusandhan (deemed to be) University, Odisha, India. He has together more than 11 years of teaching, mentoring and research experience. His research interest lies in the field of artificial intelligence, machine learning, financial engineering, bioinformatics, data mining and computer vision. He is a professional member of ISTE and ACM.

Dr. Ramamani Tripathy is presently working as Assistant Professor in the Department of Master of Computer Applications at United School of Business Management in Bhubaneswar, Odisha, India. She has 10+ years of teaching, mentoring research and academic experience at USBM. She has Ph.D in Computer Science and Engineering (CS&E) from SOA University, Odisha. She is a professional member of ISTE and ACM. Her research interest lies in the field of bioinformatics, financial engineering and data mining.

A Survey on Techniques for Early Detection of Diabetic Retinopathy

D. Vanusha, B. Amutha, Siddhartha Dhar Choudhury, and Aayush Agarwal

1 Introduction

Diabetic retinopathy is an eye-related ailment, which causes damage to the blood vessels of the retina, thereby causing blurred vision and also making it difficult for the person with the issue to spot colors. Retina is an important portion of the eye, which covers the posterior portion of the eye, which is an area sensitive to light. The main task of the retina is to convert the light that hits the eye and transforms it as signals to the brain. On progression it can lead to vision loss. This can be prevented by manual tests and manual screening. There are retinal surgeries that can treat this. Controlling the severity of diabetes and managing the symptoms, which show at early stages, can help the patient, who reached a stage where vision loss could happen. Fluorescein angiography and optical coherence tomography are two ways to diagnose DR. The former is which a fluorescein dye is injected in the eye (vein) and

D. Vanusha (✉)
SRM Institute of Science and Technology, Kattankulathur, India
e-mail: vanushad@srmist.edu.in

B. Amutha
Department of Computer Science and Engineering, SRM Institute of Science & Technology, Chennai, Tamil Nadu, India
e-mail: amuthab@srmist.edu.in

S. D. Choudhury
Software Engineer, Avalara, Seattle, WA, USA
e-mail: siddhartha.dharchoudhury@avalara.com; Aa4290@srmist.edu.in

A. Agarwal
SRM Institute of Science and Technology, Kattankulathur, India
e-mail: Aa4290@srmist.edu.in

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_11

pictures are taken which clearly shows the leaking blood vessels, while the latter is an image scan of the retina.

The idea behind this survey paper is to analyze the various trends that existed in early detection of diabetic retinopathy from traditional machine learning algorithms to current deep learning techniques. Advancement in recent technology has made it possible to overcome the overhead of manual screening and complications like applying dye in the eye, which can cause irritation or stinging to the patient. Machine learning algorithms have made it possible to classify the different classes of DR, due to the availability of large publicly available dataset in Kaggle, MESSIDOR, and DIABETRET. The main drawback with ML algorithm is feature extraction, which takes more time to extract the features. Deep learning model which tries to replicate the human brain in analyzing the objects came as a solution to fill this void created by ML algorithms, by state-of-the-art CNN models like Inception v3, ResNet50, VGG16, etc. These models have given the results in par with human grading.

Traditional computer vision algorithms are prior to deep learning algorithms. Though deep learning has more attracting features these days, traditional computer vision algorithms cannot be taken in a lighter sense, since they gave the significant results even when features had to be manually extracted and when the time taken was more to extract the features. Deep learning has a peer to peer processing, which is not the case with computer vision algorithms. To achieve the aim of prediction or classification in computer vision algorithm, the main objectives are detecting colors, edges, and corners. The human-engineered features have great significance directly on the accuracy of the prediction or classification task. So the reliability of the model is based on the human-engineered features. The traditional vision algorithms like "SURF (speeded-up robust features), SIFT (scale-invariant feature transform)," and BRIEF (binary robust independent elementary features) have significant role in extracting the features from the raw and unprocessed image. The main problem with such algorithms is difficulty in choosing feature while classifying, from the actual image. When the number of classes increases during classifica-

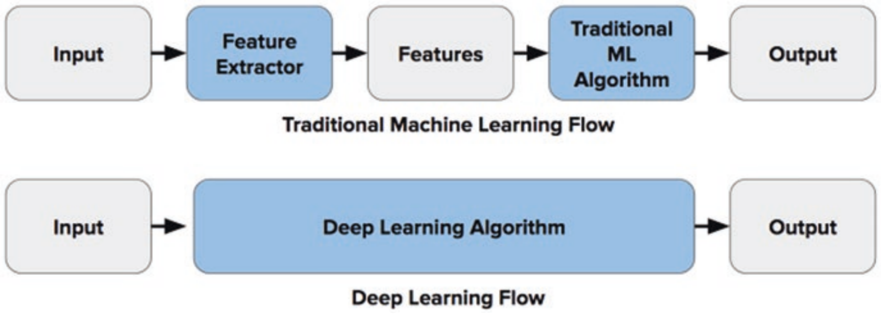


Fig. 1 Traditional computer vision workflow and deep learning algorithm workflow. *Enabling the Deep Learning Revolution.* https://www.kdnuggets.com/2019/12/enabling-deep-learning-revolution.html

tion task or when the clarity of the image is compromised, then traditional vision algorithms will got give convincing output. Figure 1 depicts the workflow of traditional algorithm and deep learning algorithm.

Diabetic retinopathy has four different stages:

1. Mild non-proliferative retinopathy: Formation of balloon-like, micro-aneurysms occur at this stage in the blood vessels of retina.
2. Moderate non-proliferative retinopathy: At this stage few blood vessels that connect to the retina are blocked.
3. Severe non-proliferative retinopathy: At this stage, many more blood vessels are blocked, which completely deprive several areas of the retina with their blood supply.
4. Proliferative retinopathy: This is considered to be the advanced stage and critical stage of DR, in which formation of new tissues occurs along the retina. They are fragile and when they start leaking, it may lead to vision loss.

Figure 2 represents the different stages of DR, micro-aneurysms (MA), exudates, and cotton wool spots.

Figure 3 shows the difference between the normal retina and DR-affected retina. Onset of MA indicates the primary stage of onset of the disease, which is a small balloon-like pouch that can leak and cause hemorrhages while the disease progresses. To analyze the image in a pixel level, segmentation is involved, which simplifies the image to a greater extent. Segmentation includes methods such as

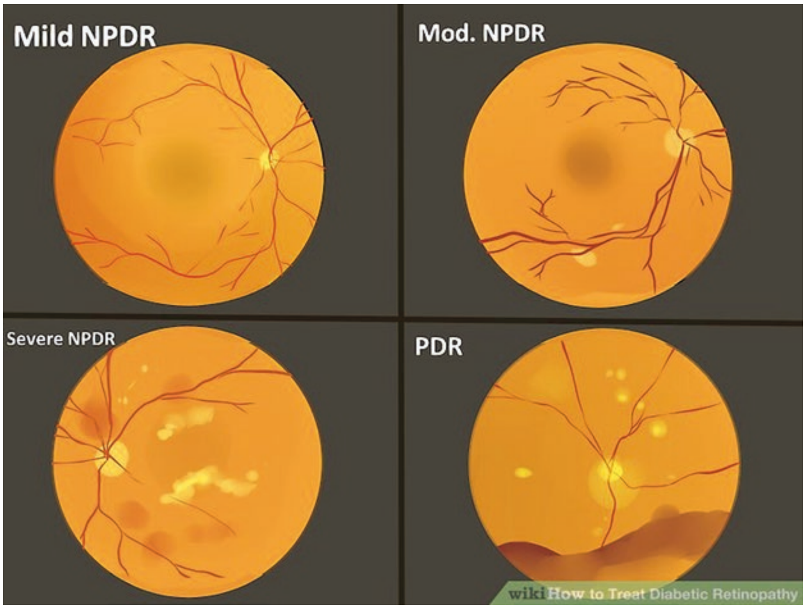


Fig. 2 Different stages of diabetic retinopathy. *How to treat Diabetic Retinopathy.* https://www.wikihow.com/Treat-Diabetic-Retinopathy

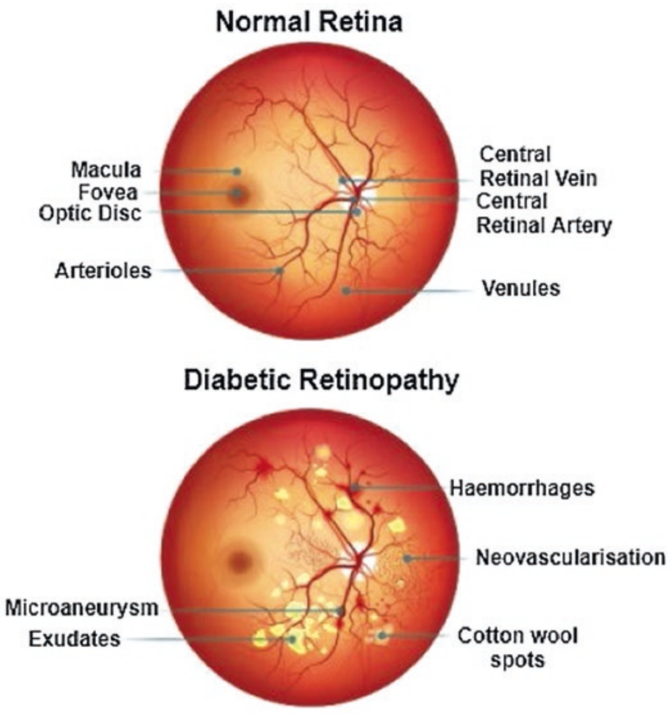


Fig. 3 Normal retina and affected retina. Investigation of machine learning methodologies in micro-aneurysm discernment. (https://www.researchgate.net/publication/338418132_Investigation_of_Machine_Learning_Methodologies_in_Microaneurysms_Discernment)

thresholding, edge detection, watershed technique, partial differential equation, and clustering-based method. Any of these segmentation techniques can be applied in medical image diagnosis like DR in pixel level to get significant results.

"The workflow of this paper is structured in the later section as follows:

- The traditional computer vision algorithms for detection of DR
- Deep learning approaches for classification of DR
- Deep learning approaches for detecting lesions in DR using CNN-based object detection models
- Deep learning approaches for DR detection using segmentation"

2 Literature Review

2.1 The Traditional Computer Vision Algorithms for Detection of DR

The paper by Hann C et al. [1] proposes computer vision methods are "developed to isolate and detect two of the most common DR dysfunctions—dot hemorrhages (DH) and exudates. The algorithms use specific color channels and segmentation methods to separate these DR manifestations from physiological features in digital fundus images. Information from color, morphology, and intensity gradients of the fundus photograph provides the means to detect the number of exudates and DHs, thus determining the presence of DR. The algorithms are tested on the first 100 images from a published database. The diagnostic outcome and the resulting positive and negative prediction values (PPV and NPV) are reported. The first 50 images are marked with specialist determined ground truth for each individual exudate and/or DH, which are also compared to algorithm identification. Exudate identification had 96.7% sensitivity and 94.9% specificity for diagnosis (PPV = 97%, NPV = 95%). Dot hemorrhage identification had 98.7% sensitivity and 100% specificity (PPV = 100%, NPV = 96%). Greater than 95% of ground truth identified exudates, and dot hemorrhages were found by the algorithm in the marked first 50 images, with less than 0.5% false positives. The improvement in this paper with the already existing methods is the use of color channels to identify DR lesions which directly allows clinical expertise and observation to be incorporated directly into the algorithm, providing a potentially far superior result.

Exudates can be found only through its high gray level variation, and the morphological reconstruction techniques help in extracting contours. Detecting the optic disc is a challenging task. The paper [2] by Walter et al. "detects the optic disc by using morphological filtering techniques and another unique technique popularly known as the watershed transformation. The algorithm is tested on a small image database, and a mean sensitivity of 92.8% and a mean predictive value of 92.4%. are achieved. The robustness feature is ensured with respect to changes of the parameters. The presence of exudates is verified by both the algorithm and the human grader.

There are four categories in this algorithm, first being image enhancement process, followed by correcting shade, normalizing image. The second step being detecting patterns, which corresponds to MA's which is made possible through the diameter closing and "automatic threshold scheme. The third step being feature extraction which finally leads to the fourth step which is automatic classification of MA, which" is possible through kernel density estimation.

In the paper [3] by Yun W.L et al., using 124 retinal pictures, the different stages of diabetic retinopathy are studied and classified under four groups, namely, "normal retina, moderate non-proliferative diabetic retinopathy, severe non-proliferative diabetic retinopathy, and proliferative diabetic retinopathy. Classification among these four classes is achieved through a fully connected three-layer neural network. "Using image processing techniques, the features are extracted from the" raw input, and it is fed as input to the classifier. Sensitivity achieved in this is more than 90% and specificity is 100%.

This work [4] by Hann et al. presents a work in which hard exudates and dot hemorrhages are graded using computer vision techniques. It takes 100 digital fundus images. In this case the specificity and sensitivity are from 95% to 100%. The reason behind achieving such good sensitivity is due to contour-finding method which combines red and green channels. This also uses the image gradient technique after applying median filter.

2.2 Deep Learning Approaches for Detection of DR

The paper [5] by Tymchenko et al. briefs the automatic detection in diabetic retinopathy, applying deep learning with a single fundus image. It also insists the involvement of transfer learning approach, which gives standard weights and bias for the network, which gives significant results in the past. Sensitivity and specificity achieved is 0.99, and this method scores a rank of 54 among 2943 methods. On APTOS 2019 dataset, it has achieved 0.925466 kappa score, which comprises of 13,000 images. To reduce the correlations between the meta features and image, data augmentation techniques are used. Three CNN architectures EfficientNet-B5, SEResNeXt50, and EfficientNet-B4, were ensembled to get QWK score 0.818462/0.924746. The enhancement of this work to the previous work is that it increases generalization and diminished the variance by using an ensemble of networks, which is pertained on a large dataset.

Generally, the process of CNN is considered as a black box. Hence this work [6] by Wang et al. proposes an interpretable DR detection, hence mitigates the black box effect of CNN. Interpretability is achieved by the user with the inclusion of regression activation map. This method helps to locate the region of interest. Activation maps are the visual representation of the activation numbers at various layers of a neural network, as the given image progresses through different layers as a result of various linear algebraic operations. In this paper, the CNN built is designed in such a way there is absence of fully connected layer; it has the convolution layer and pooling layer only. Regression activation map (RAM) provides the contribution score of each pixel in the input image. This RAM score mitigates the interpretability of CNN and makes it transparent for the one who tries to track the cause of the disease. The main idea of this method is:

1. Resampling is applied to all the classes so that classes are represented equally.
2. To initialize weights and biases, orthogonal initialization is used. It trains the smaller network on 128 pixel images and then initializes medium networks on 256 pixel images and large networks on 512 pixel images.
3. Data augmentation is used to get the input image in different angles.
4. Feature blending. The networks achieved kappa score of 0.70 with 256 pixel images and kappa score of 0.80 for 512 pixel images, and for 768 pixel images,

the kappa score achieved is 0.81. Contribution score of individual pixel by RAM is the key advantage of the proposed network in detecting DR task.

The paper [7] by Birajdar et al. proposes a pretrained DenseNet121 network with a few changes and trained on APTOS 2019 dataset. The image data generator was utilized to play out an irregular zoom of 0.15, even a vertical flip for all pictures in the dataset. It uses the benefits of DenseNets to mitigate the vanishing gradient problem, reinforce feature propagation, empower feature reuse, and diminish the quantity of parameters. The macro-average and the weighted average for precision, recall, and f1-score were assessed for five classes. The macro-average of 0.75, 0.67, and 0.70 and the weighted average of 0.86, 0.87, and 0.86 were recorded for precision, recall, and f1-score, separately. The quadratic weighted kappa files for a multi-label classification technique were assessed over validation data and found to be 91.96%. The training accuracy and validation accuracy for the single-label technique was seen as 95.98% and 94.44%, individually. However, the training accuracy and validation accuracy for the multi-label technique was seen as 97.54% and 96.40%, individually. This method has the highest accuracy among conventional CNN architectures and second highest recall.

iv. The proposed framework [8] by Birajdar et al. "works on convolutional neural networks and utilizes the AlexNet where a database of human eyes affected with diabetic retinopathy is utilized for training. The database considered is Kaggle database which comprises of more than 2000 pictures which are utilized for training the model. The trained model recognizes the presence of diabetic retinopathy and further characterizes it dependent on severity. The final model gives accuracy of up to 88%. The entire picture of the eye is broken into smaller bits of 227 × 227. This transforms one entire picture into numerous smaller pictures. The network is trained uniquely of two classes that are a clean eye having no ailment and an infected eye containing lesions. This causes the system to distinguish whether a given bit of picture of an eye contains disease or not. Each piece of the picture is "distinguished, "and each part is given a particular weight. The more the weight, the higher the priority if the lesion is distinguished in that particular part of the eye. Higher weights are given to the territory close to the optic disk and the center of the eye, and the weights are diminished as we move away from this part. This method achieves a high accuracy by using transfer learning.

2.3 Deep Learning Approaches for Detecting Lesions in DR Using CNN-Based Object Detection Models

The paper [9] by Y. Huang et al. proposes a novel way to detect hemorrhage using diabetic retinopathy images. A laudable aspect of this paper is it uses coarsely annotated fundus images which can mostly confuse neural network models. The model proposed uses a pipeline of traditional computer vision models and neural network-based object detectors to improve on the current state of the art which was held by

RetinaNet-based object detection trained on coarsely annotated fundus images. The model consists of three steps: preprocessing of image, refining coarsely annotated data using neural network, and finally using a convolutional neural network-based object detector along with a label smoothing procedure. The image preprocessing step improves the image quality and also enhances the illumination of the image. The next step is refining the coarsely annotated dataset using a novel neural network architecture called bounding box refining network (BBR-net); this helps in providing more accurate bounding box predictions in the dataset. Finally RetinaNet is trained for object detection which detects lesions in improved fundus images. They managed to achieve a better performance than pure RetinaNet-based model and achieved an exceptional IoU (Intersection over Union) score of 0.875. Pure RetinaNet-based model resulted in a top 3% mAP score of 60.9, whereas the proposed pipeline achieves 67.4 improving 6.5 mAP score points. There is a significant improvement of 11.1 on top 4% mAP and 22.9 on top 5% mAP score. Exceptional performance in detecting hemorrhage in coarsely annotated diabetic retinopathy fundus images is observed in this work.

The paper [10] by Zhang L et al. aims to detect micro-aneurysms in diabetic retinopathy fundus images. Micro-aneurysms are the earliest detectable diabetic retinopathy lesions. This paper aims at detecting these lesions using a neural network-based object detector with attention mechanism. The introduction of attention mechanism has shown significant improvement in the micro-aneurysm detection results due to the ability of attention to focus on important features in a feature map during detection. This technique first improves the quality of image using traditional image processing techniques like equalization operations. Then attention is applied to various feature maps so that important features which lead to correct prediction are extracted and fused together; these are then passed on to the micro-aneurysm detection neural network which is responsible for creating bounding boxes around the lesions. They trained their model on IDRiD_VOC dataset and achieved a sensitivity score of 0.868; this is quite important as sensitivity defines the number of true positives (a person has diabetic retinopathy and is detected to have DR) and can be used in production systems for initial screening of diabetic retinopathy before the fundus image goes to an actual human ophthalmologist. They also presented with a precision of 0.874 with 0.4 confidence, 0.885 with 0.6 confidence, and 0.895 with 0.8 confidence. They could achieve this using attention, by focusing on different parts of the feature maps while predicting the best bounding box value. Exceptional performance in detection of micro-aneurysm thus helps in early detection of DR.

The paper [11] by Chen Q et al. uses neural network-based object detector for detection of lesions in fundus images to predict diabetic retinopathy. The researchers state that the primary obstacle in detection of lesions is that the ratio of size of lesions to that of the image is extremely small which makes it tough for some common object detection algorithms like YOLO and SSD and perform poorly on this type of dataset. They propose an improved version of feature pyramid networks (FPNs) which are the backbone of models like Faster R-CNN and RetinaNet (these algorithms are specialized in detecting small objects in an image). They state that their improved technique called large-size feature pyramid network (LFPN) produces significant improvement over existing FPNs (feature pyramid networks). This model is basically a deeper FPN which allows the model to discover more feature maps and focus on smaller portions of the image. They managed to achieve 93.01% accuracy when center focus proposal strategy was used with their proposed LFPN which is the current state of the art in blot hemorrhage detection. They also displayed state-of-the-art results in micro-aneurysm detection (91.96% with LFPN and no CF proposals), hard exudate (93.79% with LFPN and CF proposals), and cotton wool spot detection (79.73% with LFPN with CF proposals). Thus with their model they set new state-of-the-art results in four of the most important techniques for early detection of diabetic retinopathy which can help ophthalmologist to check large amounts of data faster and improving chances of people getting cured of this disease. New state of the art in blot hemorrhage, micro-aneurysms, hard exudate, and cotton wool spot detection tasks for predicting diabetic retinopathy.

The paper [12] by Chen Q et al. aims to detect lesions in fundus images for predicting diabetic retinopathy. The dataset used here is not fully labelled and is thus referred to as a pseudo-labelled dataset, thus making the problem a semi-supervised learning task. The paper describes two major obstacles: (1) the dataset is not completely labelled, and (2) the ratio of size of image to the entire fundus image is very small. In this work feature proposal networks (FPNs) have been used to detect these small-sized lesions in a fundus image. The first step in this pipeline is to provide labels for the unlabeled data points using unsupervised techniques such as clustering which finds similarities between data points and clusters them accordingly to their corresponding predicted class. The whole supervised dataset which is now obtained is passed on to the FPN for detecting of lesions. With a deep feature proposal network (DFPN) and multi-round (MR) training, they achieved state of the art results of 93.04% accuracy with a confidence threshold of 0.1 in detection of lesions in the fundus images. Improvement in this work is training on pseudo-labelled and unlabeled images.

2.4 *Deep Learning Approaches for DR Detection Using Segmentation*

The paper [13] by Saha et al. describes an automated segmentation of fundus images, retina, and optic disk using CNN. The "trainable segmentation framework consists of an encoder and a decoder which is further followed by the process of pixel-wise classification, which encourages to segment" the different lesions such as exudates (both hard and soft), hemorrhages, and micro-aneurysms. Furthermore, the network in this paper is trained using binary cross-entropy, and it has utilized sigmoid activation in the last layer. Also to boost the response by a single class, softMax layer is used. The performance metrics used are sensitivity, accuracy, and positive prediction value (ppv). The optic disk's position is located using segmented output map. The previous experiments couldn't distinguish between optic disk and exudates due to lack of global view. The main objective of this work is to capture the global features of the image since the brightness level of the optic disk and hard exudates is the same, which makes it difficult to recognize them when just the local features are considered.

The paper [14] by Nayak C et al. uses two unique techniques such as multilayered thresholding technique and curvelet transform for vessel segmentation. Curvelets being a technique for multi-scale representation of object has features such as anisotropy scaling law and another important feature which is directionality. Due to these two features, edges are represented in a more efficient way than the traditional wavelets. Being a non-adaptive technique, curvelets gain its importance over traditional wavelets in fields like image processing and scientific computing. Morphological operation is applied for vessel segmentation. Additionally different threshold values are applied, and the successive layer tracking of blood vessels iteratively is done. Sensitivity and specificity are the metrics used to evaluate with clinical results. Image resolution is higher than traditional wavelet transform in this work.

Automatic segmentation can speed up the diagnosis and helps the ophthalmologist by defining a region of interest and makes the grading process easier. The paper [15] by Sambyal N et al. briefs the modified UNet architecture, which is based on residual network, which shuffles the sub-pixels periodically which are convoluted to the neighbor which is nearby. Validation and training "for micro-aneurysm and hard exudate segmentation for IDRiD dataset gives accuracy of 99.88%, sensitivity of 99.85%, specificity of 99.95%, and dice score of 0.998%. When trained on e-ophtha dataset, the model shows accuracy of 99.98%, sensitivity of 99.88%, specificity of 99.89%, and dice score of 0.9998 for micro-aneurysm segmentation. ResNet34, a pretrained model, is used as the encoder. "The residual network minimizes the degradation problem. Also, a new up-sampling technique shuffles the sub-pixel convolution which is initialized to convolution nearest neighbor resize. The proposed work obtains state-of-the-art results for the semantic segmentation of DR lesions when compared to existing work, which" is ResNet18 model.

The success of the deep learning models such as CNN is based on the large dataset. When the size of the dataset becomes small, these models are prone to large domain shift. To address this problem, the paper [16] by Bahdanau D et al. uses an end-to-end medical image segmentation model, progressive adversarial semantic segmentation (PASS), which helps to improve segmentation predictions even without requiring any domain-specific data during the training time. Domain shift problem is a serious issue which is prone to error while predicting. Domain shift problem occurs when the data from one source fails to segment properly, due to lack of generalization. In both in-domain and cross-domain evaluations, even with smaller sample size and larger domain shift, PASS performs well.

The paper [17] by Luong M.T et al. uses transformers along with convolutional neural networks for object detection tasks, and they achieve state-of-the-art results in object detection. This model has two different neural networks in it—a convolutional neural network and a transformer (encoder/decoder). Instead of fully convolutional network or adding fully connected layers at the end for object detection, they decided to use transformers which are based on the concept of multi-headed

Table 1 A comparative analysis on all categories: algorithm used, dataset, and results under different category

S. no	Algorithm used	Dataset	Category (computer vision, object detection, segmentation)	Results
[1]	Direct computer vision approach	Diaret dbO_v_1_1 database	Traditional computer vision	96.7% sensitivity and 94.9% specificity for diagnosis (PPV = 97%, NPV = 95%)
[2]	Morphological filtering techniques	30 images 640,480 digital images taken with a SONY color video 3CCD camera on a Topcon TRC 50 IA retinograph (there is no dataset used)	Traditional computer vision	Sensitivity of 92.8% and a mean predictive value of 92.4%
[3]	Classification algorithm	124 retinal images	Traditional computer vision	Sensitivity achieved in this is more than 90% and specificity is 100%
[4]	Contour-finding method and image gradient technique	100 retinal images	Traditional computer vision	Specificity and sensitivity are from 95% to 100%
[5]	Three CNN architectures EfficientNet-B5, SEResNeXt50, EfficientNet-B4	Kaggle diabetic retinopathy detection challenge 2015, IDRiD, MESSIDOR, evaluation on APTOS2019	Classification of DR	Sensitivity and specificity achieved is 0.99, kappa score of 0.925466
[6]	Regression activation map (RAM)	Kaggle diabetic retinopathy detection data	Classification of DR	For 256 pixel images, kappa score is 0.70, 0.80 kappa score for 512 pixel images, and 0.81 score is achieved for 768 pixel images
[7]	Pretrained DenseNet121 network	APTOS 2019	Classification of DR	The training accuracy and validation accuracy for the multi-label technique was seen as 97.54% and 96.40%, and training accuracy and validation accuracy for the single-label technique was seen as 95.98% and 94.44%, individually

(continued)

Table 1 (continued)

S. no	Algorithm used	Dataset	Category (computer vision, object detection, segmentation)	Results
[8]	AlexNet	Kaggle diabetic retinopathy data	Classification of DR	Final model gives accuracy of up to 88%. For more than 2000 images
[9]	Uses a pipeline of traditional computer vision models and neural network-based object detectors rather than pure RetinaNet for object detection	IDRiD, private dataset not released for public usage	Detection of DR	Pure RetinaNet-based model resulted in a top 3% mAP score of 60.9, whereas the proposed pipeline achieves 67.4 improving 6.5 mAP score points
[10]	Introduction of attention mechanism has shown significant improvement in the micro-aneurysm detection results	IDRiD_VOC	Detection of DR	A sensitivity score of 0.868, also precision of 0.874 with 0.4 confidence, 0.885 with 0.6 confidence, and 0.895 with 0.8 confidence
[11]	Large-size feature pyramid network (LFPN) produces significant improvement over existing FPNs (feature pyramid networks)	Private dataset not released for public usage	Detection of DR	93.01% accuracy
[12]	Feature proposal networks (FPNs) have been used	Private dataset not released for public usage	Detection of DR	93.04% accuracy with a confidence threshold of 0.1 in detection of lesions in the fundus images
[13]	Segmentation framework consists of an encoder and a decoder followed by the process of pixel-wise classification	IDRiD, Drishti-GS	Segmentation in DR	Sensitivity, accuracy, positive prediction value (ppv)
[14]	Multilayered thresholding technique and curvelet transform for vessel segmentation	DRIVE and STARE database	Segmentation in DR	Nil

(continued)

Table 1 (continued)

S. no	Algorithm used	Dataset	Category (computer vision, object detection, segmentation)	Results
[15]	ResNet34, a pretrained model, is used as the encoder	e-ophtha and IDRiD dataset	Segmentation in DR	"For IDRiD dataset, the network obtains 99.88% accuracy, 99.85% sensitivity, 99.95% specificity, and dice score of 0.9998 for both micro-aneurysm and exudate segmentation When trained on e-ophtha and validated on IDRiD dataset, the network shows 99.98% accuracy, 99.88% sensitivity, 99.89% specificity, and dice score of 0.9998 for micro-aneurysm segmentation"
[16]	Progressive adversarial semantic segmentation (PASS)	ARIA and CHASE datasets	Segmentation in DR	Unsupervised classification on SVHN (55%) and MNIST (98.7%)"

attention which allows the object to focus on parts of image while creating bounding boxes around the lesions. First the image is passed through a convolutional neural network with a series of convolution and pooling layers at the end of which the feature maps are flattened and passed into an encoder decoder transformer. Since transformers allow high parallelization, this model can be used for real-time object detection which is extremely important for a problem like diabetic retinopathy (Table 1).

2.5 *Inference*

After conducting the survey on various methods for early detection of diabetic retinopathy, we could infer that these three works provided the best model for this task: [7] Birajdar et al. achieved the highest accuracy in both multi-label and single-label settings, [11] Chen Q et al. produced promising results though on a private dataset, and [10] Zhang, L et al. not only provided accuracy metric which might be misleading in certain cases but also provided metrics like sensitivity and specificity which clearly shows the superiority of their model [20]; this is in part due to the usage of attention mechanism. In our future work, we would be focusing on capsule networks and transformer-based models to solve this problem [21–23].

3 Conclusion and Future Work

In this section we will discuss about some of the techniques that we believe can be useful in our own work in coming up with a solution for diabetic retinopathy detection. We discuss mainly about two recent advancements to the field of deep learning which are found to provide state-of-the-art results in computer vision and other artificial intelligence problems in general. The first of the two algorithms is called "transformers." Since the remaining technique depends in some way on transformers, this would be the first technique to be familiarized with. Transformers were first introduced in 2017 in the paper "Attention is all you need" by researchers at Google. Transformers are a way of solving problems that deal with temporal relationships in the dataset, which was until then accomplished using "recurrent neural networks (RNNs) or variants of it like" long short-term memory (LTSM) and gated recurrent units. This work solved a major shortcoming of RNN family of neural networks which is their inability to process these data with temporal dependencies in parallel. Transformers use a technique called attention along with multilayered perceptrons (MLPs) to solve these problems in parallel. Since most of the recent advances in computer vision use transformer model in their architecture, we believe that it is worth exploring in the context of diabetic retinopathy. With the help of attention mechanism (which is part of the transformer model), a model can learn an efficient way for detecting or segmenting lesions in fundus images by taking into context the neighboring pixels which can help the neural network to perform well in real-world datasets for diabetic retinopathy. The second method that we believe would be useful and worth exploring in context of early detection of diabetic retinopathy is "capsule networks." Capsule networks are a recent development in the field of computer vision which proposes that image features be represented as capsules—an entity proposed in the paper Stacked Capsule Autoencoders. This primary idea was taken from computer graphics: along with the features of an object in the capsule, it should also take into account the direction vector of that object in the image—doing this can help the system to be spatially invariant unlike the current option (convolutional neural networks). Thus capsules provide a better representation of an image and is much less susceptible to errors due to change in viewpoint. In an area (medical imaging) where accuracy is of utmost importance, introducing capsules will lead to major breakthrough in the system, thus producing and minimizing the misclassifications caused due to human errors. These use a technique called set transformers which is an improvement over the vanilla transformer model in context of computer vision. To conclude, our work will mostly focus on capsule network-based classification, detection, and segmentation of lesions in fundus images; we will also be focusing on coming up with newer transformer-based architectures for solving this particular problem and for computer vision problems in general. This will help us focus on attention-based neural networks for solving this particular problem which have shown promising results in the past.

References

1. Hann, C., Revie, J., Hewett, D., Chase, J., Shaw, G.: Screening for diabetic retinopathy using computer vision and physiological markers. J. Diabetes Sci. Technol. **3**(4), 819–834 (2009). https://doi.org/10.1177/193229680900300431
2. Walter, T., Klein, J.C., Massin, P., Erginay, A.: A contribution of image processing to the diagnosis of diabetic retinopathy-detection of exudates in color fundus images of the human retina. IEEE Trans. Med. Imaging. **21**(10), 1236–1243 (2002)
3. Yun, W.L., Acharya, U.R., Venkatesh, Y.V., Chee, C., Min, L.C., Ng, E.Y.K.: Identification of different stages of diabetic retinopathy using retinal optical images. Inf. Sci. **178**(1), 106–121 (2008)
4. Hann, C.E., Chase, J.G., Revie, J.A., Hewett, D., Shaw, G.M.: Diabetic retinopathy screening using computer vision. IFAC Proc. Vol. **42**(12), 298–303 (2009)
5. Tymchenko, B., Marchenko, P., Spodarets, D.: Deep learning approach to diabetic retinopathy detection. arXiv preprint arXiv, 2003.02261.DIABETIC RETINOPATHY (2020)
6. Wang, Z., Yang, J.: Diabetic retinopathy detection via deep convolutional networks for discriminative localization and visual explanation. In: Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence (2018, June)
7. Chaturvedi, S.S., Gupta, K., Ninawe, V., Prasad, P.S.: Automated diabetic retinopathy grading using deep convolutional neural network. arXiv preprint arXiv, 2004.06334 (2020)
8. Udayan & Gadhave, Sanket & Chikodikar, Shreyas & Dadhich, Shubham &Chiwhane, Shwetambari: Detection and classification of diabetic retinopathy using AlexNet architecture of convolutional neural networks. (2020). https://doi.org/10.1007/978-981-15-0790-8_25
9. Huang, Y., et al.: Automated hemorrhage detection from coarsely annotated fundus images in diabetic retinopathy. In: 2020 IEEE 17th International Symposium on Biomedical Imaging (ISBI), Iowa City, IA, USA, pp. 1369–1372 (2020). https://doi.org/10.1109/ISBI45749.2020.9098319
10. Zhang, L., Feng, S., Duan, G., Li, Y., Liu, G.: Detection of microaneurysms in fundus images based on an attention mechanism. Genes. **10**(10), 817 (2019)
11. Chen, Q., Sun, X., Zhang, N., Cao, Y., Liu, B.: Mini lesions detection on diabetic retinopathy images via large scale CNN features. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI), pp. 348–352. IEEE (2019, November)
12. Chen, Q., Liu, P., Ni, J., Cao, Y., Liu, B., Zhang, H.: Pseudo-labeling for small lesion detection on diabetic retinopathy images. arXiv preprint arXiv. **2003**, 12040 (2020)
13. Saha, O., Sathish, R., Sheet, D.: Fully convolutional neural network for semantic segmentation of anatomical structure and pathologies in colour fundus images associated with diabetic retinopathy. arXiv preprint arXiv, 1902.03122 (2019)
14. Nayak, C., Kaur, L.: Retinal blood vessel segmentation for diabetic retinopathy using multilayered thresholding. Int. J. Sci. Res. (IJSR). **4**(6), 1520–1526 (2015)
15. Sambyal, N., Saini, P., Syal, R., Gupta, V.: Modified U-net architecture for semantic segmentation of diabetic retinopathy images. Biocybern. Biomed. Eng. **40**, 1094–1109 (2020)
16. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv. **1409**, 0473 (2014)
17. Luong, M.T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. arXiv preprint arXiv, 1508.04025 (2015)
18. Kosiorek, A., Sabour, S., Teh, Y.W., Hinton, G.E.: Stacked capsule autoencoders. In: Advances in Neural Information Processing Systems, pp. 15512–15522 (2019)
19. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. arXiv 2017. arXiv preprint arXiv, 1706.03762 (2017)

20. Prakash, K.B., Ruwali, A. Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1
21. JHA, A.K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2
22. Kanagachidambaresan, G.R., Mahima, V., Prakash, K.B.: Programming tensor flow with single board computers, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_12
23. Kanagachidambaresan, G.R. Manohar Vinothana G.:, Visualisations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3

Correction to: Keras and TensorFlow: A Hands-On Experience

Ferdin Joe John Joseph, Sarayut Nonsiri, and Annop Monsakul

Correction to: Chapter 4 in: K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing, https://doi.org/10.1007/978-3-030-66519-7_4

References mentioned below were inadvertently added in the chapter by the volume editors. This has now been removed from the chapter.

8. Prakash, K.B., Ruwali, A., Kanagachidambaresan, G.R.: Introduction to tensor flow, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_1

9. JHA, A. K., Ruwali, A., Prakash, K.B., Kanagachidambaresan, G.R.: Tensor flow basics, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_2

10. Kanagachidambaresan, G.R., Manohar Vinoothna, G., Prakash, K.B.: Visualizations, programming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_3

11. Prakash, K.B., Kumar, A.J.S., Kanagachidambaresan, G.R.: Chatbot, pprogramming with tensor flow, EIA/Springer innovations in communication and computing. https://doi.org/10.1007/978-3-030-57077-4_9

The updated online version of this chapter can be found at https://doi.org/10.1007/978-3-030-66519-7_4

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7_12

Index

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
K. B. Prakash et al. (eds.), *Advanced Deep Learning for Engineers and Scientists*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-66519-7